



P R E L I M I N A R Y E D I T I O N

PROGRAMMING LOGIC

NAVIN MADRAS

Preliminary Edition Notice

You have been selected to receive a copy of this book in the form of a preliminary edition. A preliminary edition is used in a classroom setting to test the overall value of a book's content and its effectiveness in a practical course prior to its formal publication on the national market.

As you use this text in your course, please share any and all feedback regarding the volume with your professor. Your comments on this text will allow the author to further develop the content of the book, so we can ensure it will be a useful and informative classroom tool for students in universities across the nation and around the globe. If you find the material is challenging to understand, or could be expanded to improve the usefulness of the text, it is important for us to know. If you have any suggestions for improving the material contained in the book or the way it is presented, we encourage you to share your thoughts.

This text is not available in wide release on the market, as it is actively being prepared for formal publication. Accordingly, the book is offered to you at a discounted price to reflect its preliminary status.

If you would like to provide notes directly to the publisher, you may contact us by e-mailing studentreviews@cognella.com. Please include the book's title, author, and 7-digit SKU reference number (found below the barcode on the back cover of the book) in the body of your message.

Programming Logic

Preliminary Edition

Navin Madras

Elizabethtown Community College



Bassim Hamadeh, CEO and Publisher
Christina Brown, Acquisitions Editor
Kaela Martin, Senior Project Editor
Monica O’Keefe, Editorial Assistant
Susana Christie, Senior Developmental Editor
Abbey Hastings, Senior Production Editor
Emely Villavicencio, Senior Graphic Designer
Laura Duncan, Licensing Specialist
Natalie Piccotti, Director of Marketing
Kassie Graves, Senior Vice President, Editorial
Alia Bales, Director, Project Editorial and Production

Copyright © 2026 by Cognella, Inc. All rights reserved. No part of this publication may be reprinted, reproduced, transmitted, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information retrieval system without the written permission of Cognella, Inc. For inquiries regarding permissions, translations, foreign rights, audio rights, and any other forms of reproduction, please contact the Cognella Licensing Department at rights@cognella.com.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Cover image: Copyright © 2024 iStockphoto LP/narvo vexar.

Printed in the United States of America.



Contents

Chapter 1: Fundamentals of Programming	1
Introduction to the Chapter	1
Learning Outcomes.....	1
Section 1: Defining computers and programming	1
Section 2: The AJANRO Programming Language	3
Our first AJANRO keywords	3
Some Basics of Program Structure	4
Our first AJANRO statements	5
List of Key Takeaways.....	9
Chapter 2: Data Types, Variables, Constants and Arithmetic Operators	11
Introduction to the Chapter	11
Learning Outcomes.....	11
Section 1: Variables	11
Declaring variables	12
Section 2: Constants.....	15
Declaring constants.....	15
Section 3: Arithmetic Operators.....	16
Chapter 3: Selection Structures.....	19
Introduction to the Chapter	19
Learning Outcomes.....	19
Section 1: The Control Structures	19
The Selection Structure.....	20
Chapter 4: Iteration Structures	29
Introduction to the Chapter	29
Learning Outcomes.....	29
Section 1: The Iteration Structure.....	29
The while loop	30
Section 2: Types of Loops	33
Types of loops	33
Chapter 5: Modular Programming	41
Introduction to the Chapter	41
Learning Outcomes.....	41
Section 1: What is Modular Programming?	41
Structure of a module	42
Section 2: Benefits of Writing a modular program	45
Parameters and arguments.....	49
Flow of control.....	49
Modular program design	50
Section 3: Scope of Variables	51
Chapter 6: Arrays.....	53
Introduction to the Chapter	53
Learning Outcomes.....	53
Section 1: Understanding arrays	53
What is a data structure?	53

What is an array?	54
Section 2: Array algorithms – The Power and Benefits	59
Using loops with arrays	59
Chapter 7: Multi-dimensional Arrays	77
Introduction to the Chapter	77
Learning Outcomes	77
Section 1: Understanding multi-dimensional arrays	77
What is a multi-dimensional array?	77
Section 2: Array algorithms	80
Using loops with two-dimensional arrays	80
Section 3: Beyond two dimensions	87
Chapter 8: File Input and Output	89
Introduction to the Chapter	89
Learning Outcomes	89
Section 1: Understanding file input and output.....	89
What is file input and output?	89
How do we read and write files?	90

Chapter 1

Fundamentals of Programming

Introduction to the Chapter

In this chapter, we will learn what a program is and what a programming language is. Finally, we will get an introduction to the AJANRO programming language that will be used in this textbook, and learn the process of a program's execution on a computer.

Learning Outcomes

After reading this chapter, students will be able to:

1. Define a computer.
2. Describe what a program is.
3. Describe what a programming language is.
4. List the elements of a typical programming language.
5. Write basic, non-interactive programs in the AJANRO programming language used in this textbook.

Section 1: Defining computers and programming

This is a textbook on programming logic.

So, what is programming logic? Before we try to understand that, let us first describe what a computer is, and understand what a program is.

We all use computers, in every aspect of our daily lives. You may use a desktop computer in your college classroom or computer lab, perhaps a laptop computer on which you do your homework. You also carry a little computer in your pocket or bag – your smartphone. Many devices today, from toasters, to stoves, to cars, to watches, are computerized. It is easy to say for what purposes we use computers, but how exactly do we describe a computer in a general way? Let us try to do just that – a computer is an electronic device or machine, that accepts input

data, processes it in some way to produce some output information that we want, and is also capable of storing both the input data and the output information for future use.

Given all the various ways in which we use computers, it may seem as if a computer just knows how to do all the things we expect it to. But, from our description above, it is only an electronic device or machine. So, how does it know how to do anything? The answer is that it does not actually “know” how to do anything that it does, in the way that you simply “know” how to drive your car, or operate your oven, or do any of the hundreds of things you have learned to do as a person. Instead, a computer relies on programs to tell it what to do at each step, and how.

Having understood this, we can now define what a program is: A program is a set of instructions to the computer, telling it what to do and how to do it. The same definition applies to apps that you run on your smartphone or tablet, as apps are just programs that we happen to run on those devices.

Now that we know what a program is, let us return to our original question above – what is programming logic? Woven into each of the examples above is one requirement that is not apparent until we think about it – the instructions that the program gives to the computer, logically, have to be in the order shown; if they were in some other order, the program would not work, and, in fact, often would not even make sense. Programming logic, then, is the sequence in which instructions have to be within a program, in order for the computer to accomplish the task that we want it to and produce the correct results that we want. This is what you will learn in this textbook – how to come up with the correct logical sequence of instructions to give to the computer in order to solve any given programming problem.

Knowing that a program is a set of instructions to the computer brings up a few additional questions:

1. In what language will we write these instructions? Since we normally communicate with the English language, we assume these instructions will be in English too, but, can we use any and all English words?
2. What will the format of these instructions be – sentences? paragraphs? And are there any rules that say how we can and cannot write these instructions?
3. In the examples above, sometimes we just give a simple instruction to do one specific task, but at other times, we have the computer doing something more complex, such as, making a decision about what to do next. How do we give such instructions, which require the computer to make a decision and other such complex tasks?

Let us answer these questions one at a time, in simple terms, which will give us a list of things we will learn about programs in this textbook.

1. Yes, we will use English words. But no, we cannot use any and all English words. Instead we will use a **programming language**, which contains only a small set of words from the English language. This small set of words that we can use are called the programming language’s **keywords**.

2. The instructions will be in the form of short **statements**. Yes, there are rules – just as in the full English language, where we have learned rules that tell us the correct order in which to use words so that our sentences will make sense. Likewise, programming statements have to use the keywords from that language in some proper order that the rules specify, so that those statements will make sense. This set of rules is called the programming language’s **syntax**. Also included are rules that say how to use the **operators** with which you are familiar from Math classes, for addition, subtraction and so on.
3. Every programming language also provides a set of **control structures** – as their name suggests, these are structures that control the flow of the instructions and of the program’s logic, and provide a way to specify when the computer is supposed to make a decision and choose one out of two options, and so on.

You may heard the act of writing programs referred to as “coding”. Since you use a specialized programming language as described above, and not regular English, to write instructions to the computer, you are considered to be writing those instructions in a kind of “code” that the computer understands – therefore, instructions within a program are referred to as “code”, and the process of writing them, as “coding”. From this point forward, whenever we want to say “instructions to the computer”, we will instead simply say “code”.

Section 2: The AJANRO Programming Language

In order to learn programming logic, we need a language in which to write our programs. Most programming languages that are used by professional programmers are fairly large and complex. For our purposes, all we need is a simple language that will illustrate all of the parts of a programming language that we have listed at the end of the Introduction section of this chapter. So, the author has created a simple language with the help of three former students – Tiffany “AJ” Dowell, Andrew Case and Robert Duvall. The language’s name uses the first two letters of each of their first names – AJANRO (pronounced A-Jaw-n-ro). We will use this language throughout this textbook.

Our first AJANRO keywords

In this section, we will learn our first four keywords in our AJANRO programming language. Here are the first two of them:

- `startprogam`
- `endprogram`

As their names suggest, `startprogram` simply says that this is where our program starts, and `endprogram` says this where our program ends. Note that there is no space between the words “start” and “program”, and likewise, between the words “end” and “program”. In general, we will not include any spaces within any of our keywords or within names of structures and other items within our programs.

Here is an example that shows how these first two keywords are used:

```

startprogram GreeterProgram

...

endprogram

```

The line that begins with `startprogram` says, this is where our program begins and here is the program's name, which, in this case, is `GreeterProgram`.

The line that contains simply `endprogram` says that this is where our program ends.

The instructions that our program wants to give to the computer – in other words, the code for our program – will be written in the space between those two lines.

Note that both `startprogram` and `endprogram` are simply *marker* keywords, used to mark where the program's code begins and ends, and do not indicate an action of any kind. They are not instructions to the computer to say it should start this program's execution now, and that it should now stop or end this program's execution.

Now, let us look at our next two beginning keywords – they are:

- `startmain`
- `endmain`

Here is a revised version of the example from earlier, applying these two keywords

```

startprogram GreeterProgram

    startmain

        code

    endmain

endprogram

```

As we will learn in a later chapter, programs can contain several subunits or modules that contain code to do various tasks. Every programming language specifies a way in which we can indicate which one of those subunits or modules should be the starting point for the program's execution. The module with which a program's execution begins is called its **main module** or **mainline logic**. In our AJANRO programming language, the `startmain` keyword indicates where the mainline logic begins, and `endmain` indicates where the mainline logic ends. As mentioned earlier, in a later chapter, we will see how to write parts of a program other than the mainline logic, but for now, our programs will be simple enough that we can write all code within the mainline logic itself, so, for the moment, all our code will be enclosed between the `startmain` and `endmain` keywords.

Some Basics of Program Structure

Before we go on to our next two keywords, let us take a look at the structure of our programs, to learn some common programming practices that we will follow:

1. Names of our programs, e.g., GreeterProgram, may contain more than one word, and are typed with no spaces between the words, but capitalizing the first letter of each word. Any such pattern that you use in naming programs, and items within programs, is called a **naming convention**. This particular one, where we capitalize the first letter of each word, is called the **PascalCase** naming convention.
2. Each level of the program that is enclosed within an outer level is indented by pressing the TAB key on the keyboard. In the above example, the startprogram and endprogram lines are at the left margin; startmain and endmain are enclosed between those, and are therefore indented by one TAB; the code inside the mainline logic is indented by one further TAB. This is to make it easy to spot at a glance, without having to read every line of code, where the program begins and ends, where the mainline logic begins and ends, and where the code within the mainline logic (and later each of the other modules that we write) begins and ends. Think of this as a rule similar to how you are told to indent the first line of every paragraph when writing a paper.
3. As we learn more concepts, keywords and control structures, we will learn more naming conventions and code structuring practices.

Our first AJANRO statements

Earlier, in our discussion of what a program is, we learned that program instructions are written in the form of short statements. Let us now make a technical definition of a statement:

In programming, a **statement** is a line of code that says to do some specific action, such as “get this data as input from the user” or “show this message to the user” or “do this calculation”.

The first two types of statements that we will learn are:

- the output statement
- the input statement

In connection with these, we learn our next two keywords, which are:

- input
- output

These two keywords do not actually exist in professional programming languages.

The reason for this is, on any actual computer or other device, the input may come from any of various sources:

- A keyboard, on which a human user is typing some data in response to the program’s telling them to do so
- A file on the disk

- A barcode scanner
- A microphone or camera

Similarly, a program's output may go to various destinations

- The monitor, to be viewed by a human user
- A file on the disk, to be used later
- A printer

Each one of these sources of input and destinations for output requires some set-up steps within a program, and somewhat different-looking input and output statements in each case.

Since this textbook is about learning the *logic* of programming, i.e., the logical order in which things should happen, e.g., get this input data first, do these calculations with it second, and show these results as output third, rather than with the nitty-gritty details of what the source of each input is and how to get it from there, what the destination for each output is and how to send it there, we keep our input and output statements simple. We do this by assuming that our input will always come from the keyboard, and our output will always go to the monitor, and that we do not have to do any setup steps to make either one happen. In the final chapter of this textbook, we will see one small and again highly simplified variation – input and output with files.

The output statement

Let us now dive into our first type of statement – the output statement, which uses the output keyword.

We learned earlier in our discussion of what a program is, that all programming statements in a language have rules that specify how to write them correctly, and that this set of rules is called the **syntax** for that language.

Here, then is the general syntax for writing an output statement, first stated in a sentence, followed by the abbreviated form that will be used to specify the general syntax for other types of statements in the rest of this textbook:

Sentence form:

Simply type the output keyword, followed by whatever you want to output on the monitor

Abbreviated form:

output whatever you want to output

Here is a revised version of the GreeterProgram from earlier, this time, giving a single instruction to the computer, in the classic tradition of learning programming, as generations of programming students have done in their very first computer program in any programming language – to simply output the message “Hello, World!” to the monitor's screen.

```

startprogram GreeterProgram
    startmain
        output "Hello, World!"
    endmain
endprogram

```

You can see how the single line of code within the mainline logic, the output statement, follows the general syntax for output statements shown above.

Try it:

1. Write a similar program named Introducer that outputs two different messages

- "Hello class!"
- "My name is _____" (fill in your name where the blank is)

To do this, you would write two output statements – one for each message

2. Now, pick three short lines of text from anywhere – a cereal box / a textbook for another class / dialogue from your favorite movie or show – and write a program with a name of your choice, that describes what the program does, just like the names GreeterProgram and Introducer do, containing separate output statements to output each of those.

As we learn more kinds of statements that we can write in programs, this textbook will first present the general syntax for that type of statement, followed by examples, and then have you try it out, as we did with the output statement above.

The input statement

As mentioned earlier, we assume all input is coming from the keyboard, and no setup steps are needed to make this happen.

General syntax:

input where inside the computer we want this input to go

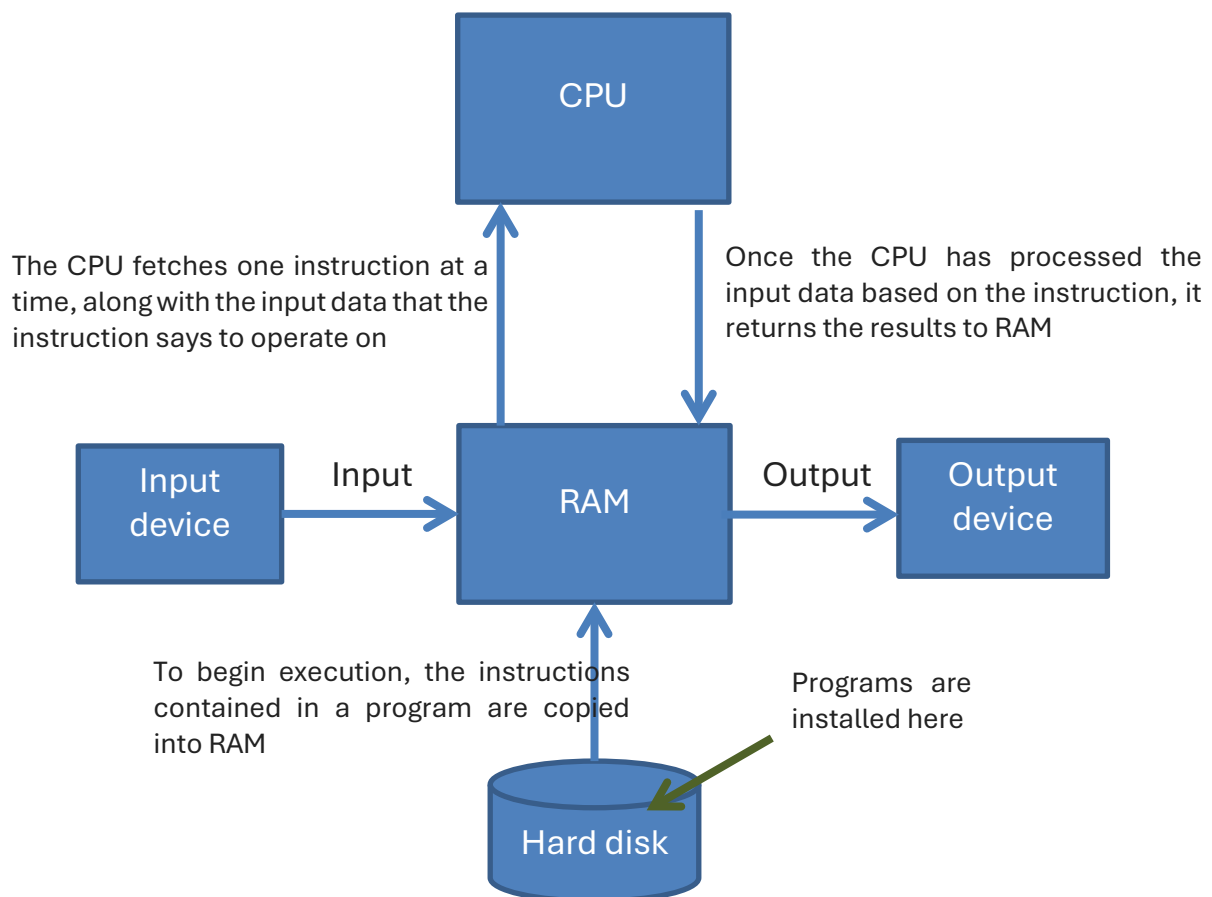
This brings us to where we have to answer this question for ourselves first: when the computer receives input data, such as, when the user types something at the keyboard, or speaks into a microphone, or captures a video with a webcam, where exactly within the computer does this input data go? This requires a little more background knowledge, which we will gain in the next section.

How programs run on a computer (or other device)

For this discussion, let us consider the sequence of events that occur inside the machine, when you run a program, such as a web browser, on a desktop or laptop computer.

1. In these types of computers, programs are installed on the hard drive.
2. You click, or double click, a program's icon to run it.
3. A copy of the program is then made and loaded into the computer's Random Access Memory (RAM).
4. The computer's Central Processing Unit (CPU, or, simply, the processor) starts fetching instructions that the program contains, from RAM, and then doing what they say.
5. Meanwhile, you, the user, are interacting with the computer, e.g., by typing input at the keyboard when it expects you to, and viewing its output on the screen.

Here is a picture that illustrates this process:



From this discussion and picture, you can see that the two internal resources of a computer that are in active use while a program is running are the CPU and RAM.

Out of these two, the CPU follows the instructions given by a program.

RAM functions as a warehouse that holds everything that the CPU requires:

1. A copy of the instructions themselves
2. Input data entered by the user
3. Output results that the CPU generated by operating on the data following the instructions given to it.

This answers the question of where all input data goes – it goes into the computer’s RAM. But then, we have a new question – in an input statement, how do we specify *where exactly within the computer’s RAM* we want this particular piece of input data to go?

Chapter 2 will answer that question, and introduce you to some further programming concepts, as well as some more AJANRO keywords and types of statements.

List of Key Takeaways

- A computer is an electronic device that follows instructions to get input data, process it, and provide output information.
- Every program is a set of instructions to the computer (specifically to its CPU) saying how to process the data to produce the desired results, i.e., output information.
- Programs are written using programming languages.
- Each programming language allows us to use certain words (keywords) and symbols (operators), following a set of rules (the language’s syntax)
- We write instructions in a program in the form of statements – lines of code that each specify some action to be performed.
- We learned the keywords `startprogram`, `endprogram`, `startmain`, `endmain`, `input` and `output` in the AJANRO programming language that will be used in this textbook.
- We learned the general syntax for output and input statements, which use the `output` and `input` keyword, respectively.
- Instructions contained in a program that is currently running on the computer, the input data it receives and the results it produces by processing that data, are all located in the computer’s RAM.

Chapter 2

Data Types, Variables, Constants and Arithmetic Operators

Introduction to the Chapter

In the previous chapter, we got an introduction to the AJANRO programming language that will be used in this textbook and learned the process of a program's execution on a computer. Now, we turn our attention to how programs use the computer's RAM to store data that they require while they are executing.

Learning Outcomes

After reading this chapter, students will be able to:

1. Describe variables and constants
2. Describe what a data type is and the data types that are available in the AJANRO programming language
3. Declare variables and constants in AJANRO

Section 1: Variables

In algebra, you learned that a variable is a quantity whose value changes, or varies, which gives us the term variable, i.e., a variable is something that varies. This definition holds in programming as well, with some additional technical aspects to it.

At the end of Chapter 1, we said that the computer's RAM holds everything that the CPU requires, including both input data entered by the user and the output results that the CPU generates by operating on the input data following the instructions given to it by a program. We can describe this in terms of what a program does as – programs put the data input by the user and the output results they generate, at various locations in the computer's RAM. In order to do this, when each program starts up, it has to set aside, or “reserve” locations in RAM that it intends to use for these purposes.

We are now ready to define a variable in programming terms – it is a location in the computer's RAM that the program has reserved so that it can put input data or results of

processing data there. Further, it is a RAM location whose contents can vary *while the program is running* – hence the name variable.

This process of reserving spots in the computer’s RAM for a program’s use is called **declaring variables**, and the lines of program code where we do this are called **declarations**.

Our next question then is: How do we go about declaring variables?

Declaring variables

In any programming language, there are two components to a variable declaration:

1. The name of the variable
2. Its data type

Let us look at each of these.

The name of a variable or the variable identifier, as it is also known, is the name by which our program will refer to a location in RAM that it intends to use. You may have learned about the structure of a computer’s RAM, i.e., that it is organized into bytes. On your own computer, for instance, you may have 4 GB of RAM, by which you understand that your computer contains about 4 billion bytes of RAM. As far as a computer’s operating system, e.g., Windows 11, is concerned, each byte of RAM is identified simply by a number – the very first byte would be byte #0, the next one would be byte #1, and so on. This is good for the operating system whose job it is to manage a computer’s resources, but for a person writing a program, it is both hard to think in terms of which byte numbers our program will use, and in fact, impossible for us to have any knowledge of what other programs may already be running on the computer when our program starts up, and which byte numbers those programs are already using and are therefore unavailable to our program. Instead, we just assign logical names to RAM locations that our program wants to use, e.g., if we want to store the price of an item at some RAM location, we could assign the name `price` to that location. Within our program’s code, that location will simply be known as `price`, while the operating system determines which exact byte in RAM that name refers to and we do not have to concern ourselves with that low-level detail.

A variable’s data-type is a specification of what kind of data we want to allow at that RAM location, i.e., do we want to allow a string of alphanumeric characters, or a whole number, or a number that could contain a fraction, or something else. A couple of simple reasons for specifying this would be

1. Our program must be able to rely on the RAM locations that it is using, to contain the kind of data that it expects there, e.g., if our program expects to find a number that it can use in some calculation, at some particular location, it must be able to rely upon the fact that there will be only a number there and not a string of characters.
2. Different kinds of data occupy different amounts of RAM, i.e., some may require only two bytes, some may require four bytes. So, by specifying a data-type when declaring a variable, we are also saying *how many bytes of RAM* we want to reserve for some data that we plan to store there.

Now that we understand the concept of a data-type, let us look at what data-types are supported in our AJANRO programming language.

AJANRO Data-types

Different programming languages use different names for the data-types that they support. This is just a matter of the language creator's choice – however, the *categories* of data-types that are supported are the same in all languages, i.e., they all provide a data-type for a string of characters, one or more for whole numbers only, one or more for numbers that may contain a fractional part, and so on. In our simplified language, we have just five data-types:

1. `char`: this data-type is used for single characters, e.g., a person's middle initial
2. `string`: this data-type is used for a group, or string of characters, e.g., a person's first name
3. `integer`: this data-type is used for whole numbers, e.g., the number of classes you are taking this semester, which cannot possibly contain any fractions
4. `floating`: this data-type is used numbers that could contain fractions, but don't have to, e.g., the price of some item
5. `boolean`: this data-type is used to indicate whether something is true or false

Try It!

1. Figure out which of the above data-types would be applicable for the data items listed below:
 - a. The brand name of your shoes
 - b. The price of your shoes
 - c. The number of classes in which you are enrolled this semester
 - d. Your letter grade in a class
 - e. Whether a game that you are playing is over or not
-

Writing variable declarations

Now that we know what variable names/identifiers are, what a data-type is, and what data-types are present in our language, we can learn how to write a variable declaration in our language. Here is the general syntax:

```
data-type variableName
```

This is the format of a variable declaration in most programming languages; a few of them use a slightly different format; but they all require these two components of a data-type and a variable name. Let us look at a few examples, using each of the above data-types:

```
char middleInitial  
string bookTitle  
integer seatsAvailable  
floating ticketPrice  
boolean isApproved
```

Some points to note in all of these sample declarations are:

1. The names of variables explain logically what that variable will be used for (i.e., what data will be stored at that RAM location), e.g., `bookTitle` clearly refers to the title of a book, `seatsAvailable` refers to how many seats are still available, for, say, a concert.
2. All the other variable names are simply nouns, while the very last one, `isApproved`, seems to be asking a question that can be answered “yes” or “no”, or, more accurately in programming, as `true` or `false`. This is also a common programming practice when naming variables of the `boolean` data-type (remember that this data-type is used to indicate whether something is true or false only), and we will follow it in this textbook.
3. The names are typed in what is called `camelCase` – the variable’s name begins in lower-case, and if there are any additional words in the name, the first letters of those words are capitalized. This makes it easier to read a variable’s name, as the capital letter in the middle of a name cues us that one word has ended and a new one has begun at that point. This is a common naming practice in programming, so, we will follow it in this textbook too.

Try It!

Write complete variable declarations for the data items listed below – these are the same ones for which you simply figured out data-types earlier:

1. The brand name of your shoes
 2. The price of your shoes
 3. The number of classes in which you are enrolled this semester
 4. Your letter grade in a class
 5. Whether a game that you are playing is over or not
-

Section 2: Constants

As with variables, you have learned about constants in algebra and science classes – these are quantities whose values do not change. For example, in mathematics, π is a constant with the value 3.14159 and continuing on for an endless string of digits, in physics, gravitational acceleration is a constant with the standard value of 9.80665 m/s².

The term constant in programming means much the same, with one additional detail – a constant is a quantity whose value does not change *while the program is running*.

Constants in programming are used to represent both mathematical and physical constants like the two mentioned above, but also other values that the program uses, which tend to remain the same for long periods but can change over a matter of months or years or decades. An example of this would be a sales tax rate – this is a quantity that does not change in the middle of a program's execution, but rather over some years, based on regulations that are passed, and so on. Here are some other examples of programming constants, some of which can change over time while others cannot:

1. The discount rate offered on some product
2. The maximum number of students who can enroll in a class
3. The capacity of a single storage container such as a tank
4. The number of pounds in a kilogram
5. The number of months in a year

Declaring constants

Just like variables, a program has to declare the constants that it intends to use within its code, before it can use them.

The general syntax for declaring a constant in AJANRO is similar to that for declaring variables, with an additional keyword and one additional requirement. Here it is:

```
const data-type NAME_OF_CONSTANT = value
```

The `const` keyword indicates that we are declaring a constant rather than a variable. Since a constant cannot change while the program is running, it cannot be assigned a value later – its value has to be assigned when it is declared.

Here are examples, declaring the constants described in the previous section:

```
const floating DISCOUNT_RATE = 0.05
const integer MAXIMUM_ENROLLMENT = 30
const floating TANK_CAPACITY = 17.5
const floating POUNDS_PER_KILOGRAM = 2.2
```

```
const integer MONTHS_PER_YEAR = 12
```

As we saw with names of variables, names of constants must fully explain themselves, e.g., TANK_CAPACITY clearly means the capacity of a tank, whereas simply CAPACITY would leave it unclear what we are representing the capacity of.

Try It!

Write declarations for the constants listed below, making up values for the ones that are not well-known values:

1. The number of days in a week
 2. The price of admission at a museum
 3. The maximum weight capacity of an elevator
 4. The name of the college where you are a student
 5. The seating capacity (number of seats) of a stadium
-

Section 3: Arithmetic Operators

Every programming language contains several operators, for various purposes. We will learn other types of operators in later chapters. At the moment, we will focus on only the arithmetic operators.

In programming, these are the same operators that you learned in arithmetic or algebra classes:

1. + for addition
2. – for subtraction
3. * for multiplication
4. / for division

The order of operations that you learned in algebra class applies to these in programming too:

1. Multiplication and division are performed first, from left to right
2. Addition and subtraction are performed second, from left to right
3. You can force any order of operations that you want to by using parentheses

The one additional operator in this category that you will encounter in most programming languages, and which is present in our language, is the % operator. This is called the **remainder** or **modulus** operator.

We will take a look at this operator in detail in a later section, after we first see a complete program that declares variables and constants, and uses some of the arithmetic operators that you already know.

Chapter 3

Selection Structures

Introduction to the Chapter

In the previous chapter, we learned about data-types, variables, constants and arithmetic operators, and saw examples of simple programs that apply those concepts. Now, we will learn about how programs make decisions.

Learning Outcomes

After reading this chapter, students will be able to:

1. Name the control structures that are used in programming.
2. Describe what a selection structure is, and the only type of selection structure in AJANRO – the if statement
3. List the relational operators in AJANRO
4. Describe what a Boolean expression is
5. Write and evaluate Boolean expressions
6. Write programs in AJANRO that utilize “if” statements to make decisions

Section 1: The Control Structures

The structured programming theorem describes a set of structures that can be used to solve any programming problem that has a solution – this of course means that there are some problems that cannot be solved, which we will ignore in our study of how to solve programming problems in this textbook.

The set of control structures that we require, and are supported by most programming languages, is

1. Sequence
2. Selection
3. Iteration

Let us briefly describe these before we go on to focus on selection structures, which are the topic of this chapter.

1. Sequence – this simply means that the program’s statements are executed in the order they are written. To say this as an instruction on how to write a program, we could say – we should write program statements in the order we need them to be executed in order to perform the tasks required of the program. We have applied this since the very first example program presented in this textbook.
2. Selection – here, we say that the program selects the next statement to execute, based on some condition being true. To state this in terms of programs making decisions, we could say that the program decides which statement to execute next based on some condition being true. For this reason, this is also referred to as a decision structure.
3. Iteration – this structure is used when a program has to repeatedly execute some statements so long as some condition is true. This is also called a repetition structure, or loop.

The Selection Structure

As described earlier, programs use this structure to select the next statement to execute. Programming languages usually support a few different types of selection structures. To keep things simple, in our AJANRO programming language, we have only one variety of selection structure – the if statement.

Before we take a look at what an if statement looks like in our language and how to write one, let us first think of a few instances of how we use the word “if” to state decisions or choices (i.e., selections) in plain English. You might say something like, “if it is cold, I will put on a jacket”, or “if it is 11:00am, I will go to class”, or yet, “if this bank’s interest rate is lower, I will select their loan, otherwise, I will select this other bank’s loan”. In each case, we state a condition, e.g., “it is cold” followed by what we will do when that condition is true. If statements in programming are modeled on the same decision-making process.

Here, then, is the general syntax for an if statement in our language:

```
if condition
    what we want the program to do when the condition is true
else
    what we want the program to do otherwise
endif
```

In order to be able to write if statements, we have to learn a couple of preliminary topics.

Relational Operators

In our examples of decisions stated in plain English earlier, we simply described some condition in general terms, e.g., “if it is cold”. To describe a condition in a program, however, we have to write it in some quantifiable way, e.g., “if the temperature is below 50 degrees” and express the same not in words but in terms of variables and symbols. To do this, let us assume that we have declared a variable named `temperature`, and that it contains whatever the user has input as the current temperature. We could then express “temperature below 50 degrees” as

```
temperature < 50
```

From having learned it in algebra class, you recognize the symbol “<” as meaning “less than”. In programming terms, this symbol is said to be a **relational operator**, i.e., one that tests the relationship between two quantities, which, in this example, are `temperature` and `50`. Below is a list of the relational operators in AJANRO, along with their meanings – most of them are familiar to you.

Table 3.1 Relational operators in AJANRO

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
!=	Not equal to

The first four of these generally remain the same in all programming languages; the last two can be different in different programming languages.

Boolean Expressions

In programming terms, an expression is something that has a value, e.g., `4 * 5` is an expression that has the value `20`. There are different types of expressions. An expression such as the `4 * 5` above, or `x + y`, where `x` and `y` are some two integer variables that have some two numbers stored in them, are called **arithmetic expressions** – they contain arithmetic operators such as `*` and `+` and produce some numeric result. An expression such as `“Hello “ + firstName`, where `firstName` is a string variable, would be called a **string expression** – it uses the `+` operator to combine `“Hello “` and the contents of the `firstName` variable to produce a longer string of characters.

In a similar way, the example of `temperature < 50` shown above is an expression that can have only two possible values – `true`, or `false`. Such an expression is called a **Boolean expression**. In general, any such expressions that you write, using the relational operators that we learned above, would be a Boolean expression.

Try It!

Write Boolean expressions that test the items below:

1. temperature is no more than 90
 2. pointsEarned is at least 40
 3. response is not the same as EXIT
 4. currentName is the same as or comes alphabetically after previousName
-

If statement

Having learned what a Boolean expression is, we can now come up with a more technical definition and general syntax for the if statement:

```
if Boolean expression
```

```
    what we want the program to do when the expression is true
```

```
else
```

```
    what we want the program to do otherwise
```

```
endif
```

The section of the if statement where we say what the program should do when the Boolean expression is true is called the if clause. The part below else is called the else clause.

The else clause is optional, i.e., sometimes, we may want the program to do something when the Boolean expression is true, but there may be nothing to do otherwise.

For a beginning example of an if statement, let us look at one that requires only the if clause. Suppose a clothing company offers all customers who purchase 10 or more shirts a discount rate of 5%. Assuming that the shirtsPurchased and discountRate variables have been declared with appropriate data-types, and that the user's input of the number of shirts purchased has already been stored in the shirtsPurchased variable, our if statement would look like this:

```
if shirtsPurchased >= 10
    discountRate = 0.05
endif
```

Since we were only given that a purchase of 10 or more shirts receives a discount rate of 5%, there was nothing to do when the purchase was not in that range, therefore, we simply omitted the else clause.

Now, suppose our clothing company decides to give a 5% discount rate for purchases of 10 or more shirts, and a 1% discount rate for lower quantities, then our if statement would look like the one below:

```
if shirtsPurchased >= 10
    discountRate = 0.05
else
    discountRate = 0.01
endif
```

Try It!

Write if statements that do the following:

1. If temperature is no more than 90, output a message that it is safe to go outside, otherwise, say to stay home
 2. If pointsEarned is at least 40, set group to ELITE, and otherwise, set it to STANDARD
 3. If isGameOver equals true, then output whatever playerScore is with a message, otherwise, output a message saying to play their next move
-

Combining if statements

The example so far has shown only a single if statement that selects one out of two options, i.e., if some Boolean expression is true do one thing, otherwise, do another. Most programs have to make more complex decisions, which require us to write combinations of if statements.

There are two ways in which you can combine if statements (and, as you will see later in Chapter 4, the same two ways in which you can combine iteration structures):

1. Nesting – writing one or more if statements inside another if statement
2. Stacking – writing a set of two or more if statements one below the other

Nesting if statements

Consider our clothing company again – suppose they decide that for all purchases of 10 or more shirts, they will offer a discount rate of 5%, otherwise, a discount rate of 1% only for purchases of 5 or more shirts. In this case, then, we have to apply a further test within the else clause of our if statement shown above, to test whether shirtsPurchased is 5 or more. Here is what that would look like:

```
if shirtsPurchased >= 10
```

```

        discountRate = 0.05
else
    if shirtsPurchased >= 5
        discountRate = 0.01
    endif
endif

```

The second if statement here is nested inside the first; more specifically, it is nested inside the else clause of the first.

You can nest if statements inside the if clause or the else clause of an outer if statement, or even both.

Try It!

Write if statements that do the following:

1. If temperature is more than 32, then test if it is no more than 90 and output a message that it is safe to go outside, otherwise, say to stay home – note that you will be nesting the second if statement inside the if clause of the outer one here.
 2. If pointsEarned is at least 40, set group to ELITE, otherwise, if pointsEarned is at least 30, set group to ADVANCED, and otherwise, set it to STANDARD
 3. If isGameOver equals true, then output whatever playerScore is with a message, and then if playerScore is greater than highestScore, output a message saying they have a new highest score, otherwise (i.e., if the game is not over) output a message saying to play their next move.
-

Remember, the if statement is the only tool by which our program can make decisions, so, we do whatever amount or level of nesting is necessary to produce a program that does the required tasks. Suppose our clothing company decides to offer a more complex discount rate structure:

Table 3.2

Shirts purchased	Discount
1 to 3	0%
4 to 6	1%
7 to 9	2.5%
10 or more	5%

The if statement that implements this would look like:

```
if shirtsPurchased >= 10
    discountRate = 0.05
else
    if shirtsPurchased >= 7
        discountRate = 0.025
    else
        if shirtsPurchased >= 4
            discountRate = 0.01
        else
            discountRate = 0
        endif
    endif
endif
```

This time, we have three if statements – an outer one, with one nested inside that, and the third one nested inside the second one. The program, based on which one of the three Boolean expressions is true, is going to select one out of four values – 0.05, 0.025, 0.01 or 0 – to assign to discountRate. This is an example of **multi-level selection**.

Now, let us see a complete program that applies nested if statements.

Stacking if statements

As described earlier, this refers to writing a set of if statements one below the other. Picture a stack of books – a stack of if statements would be similar.

Suppose we have an ice-cream shop that offers two simple options as follows:

1. Size – SMALL/MEDIUM/LARGE
2. Container – CUP/CONE

The shop charges different prices of \$1.99, \$3.89 and \$5.79 for the three sizes; for a cup, there is no charge, but for a cone, there is a charge of \$0.99. Our program has to determine the total for an ice-cream order. In this case, since the options of size and container are independent of each other, i.e., you can get any size you want, and any type of container you want, out of those offered, the if statements that determine the charge for the size and the charge for the container would have to be independent of each other as well – in other words, we would have a stack, of

the if statement that determines the size charge first, and below that, the one that determines the container charge.

Assuming that all of the variables and constants that you see in the code below have been declared, and we have gotten the user's input of the size and container, the if statements would look like this:

```
if size = SMALL
    sizePrice = 1.99
else
    if size = MEDIUM
        sizePrice = 3.89
    else
        if size = LARGE
            sizePrice = 5.79
        endif
    endif
endif

if container = CUP
    containerPrice = 0
else
    if container = CONE
        containerPrice = CONE_PRICE
    endif
endif

totalPrice = sizePrice + containerPrice
```

Now, let us see a complete program that applies stacked if statements.

What to include in an if statement

Now for a general rule to follow when writing if statements, to make them easier to write, read and update in the future:

In each if statement, include only the specific parts that are directly determined based on the if statement's Boolean expression. For instance, in the ice-cream shop example shown earlier, the first if statement figures out the price for the size of the ice-cream, and does nothing else, because only the size price is directly determined based on the size; likewise, the second if statement concerns itself only with figuring out the container price, since only that is determined by what kind of container the customer wants. All the code for declaring the variables and constants, having the user input what size and container they want, is all preliminary to the if statements, and is written before them. The calculation of the total price has to happen regardless of what the size and container are, and it depends on the prices for those having been figured out already, so, it comes after the if statement.

Chapter 4

Iteration Structures

Introduction to the Chapter

In Chapter 3, we learned the structured programming theorem, which tells us that the set of control structures that can be used to solve any programming problem that has a solution are those of sequence, selection, and iteration. In that chapter, we learned about selection structures in detail, after observing that we have been using the sequence structure since Chapter 1. We went on to learn the specific selection structure that is in our AJANRO pseudocode language used in this textbook and is present in just about every programming language – the if statement. Now, in Chapter 4, we will look at the last of the control structures – the iteration structure – in detail.

Learning Outcomes

After reading this chapter, students will be able to:

1. Describe what an iteration structure is, and the only type of iteration structure in AJANRO – the while loop.
2. Describe the types of loops.
3. Write programs in AJANRO that utilize while loops.

Section 1: The Iteration Structure

In Chapter 3, though we did not look at the iteration structure in detail, we saw a definition of it – that it is the structure that is used when a program has to repeatedly execute some statements so long as some condition is true. We learned that it is also called a repetition structure, or loop.

Before we see the details of this structure, let us look at a few scenarios where a program has to repeatedly do something.

1. Think of any company that sends you a bill, say, the electricity company – to generate customers' bills, the company's program would have to repeatedly perform the following tasks for each customer
 - a. Access the customer's account details

- b. Calculate how much electricity they have used
 - c. Calculate their total for that usage
 - d. Add any taxes and other fees to come up with a total amount due
- 2. A college wants to generate a report that shows, say, the total enrollment at the college. A program that does this for the college would repeatedly
 - a. Access a course's enrollment
 - b. Add that to a running total of enrollment in all courses at the college

You can certainly think of many more examples where you can figure out that a program has to repeat the same steps over and over again. Indeed, this ability to repeatedly perform the same steps in a way that is exact, without getting bored or tired by the repetition, is one of the great advantages of computers and programs.

The while loop

Programming languages usually support a few different types of iteration structures. To keep things simple again, in our AJANRO programming language, we have only one variety of iteration structure – the while loop – which is present in just about every programming language.

Before we take a look at what a while loop looks like in our language and how to write one, let us first think of a few instances of how we use the word “while” to state repetition of something (i.e., iterations) in plain English. You could describe the act of driving to some destination in a general way as, “while I have not reached my destination yet, I will continue driving”, or you could describe enrolling students in a class with a maximum capacity of 24 as “while the number of students enrolled is less than 24, we will sign students up for the class.” In each case, we state a condition, e.g., “the number of students enrolled is less than 24” followed by what we will do while (or, in other words, so long as) that condition is true. While loops in programming are modeled on the same process of performing some repetitive task.

Here, then, is the general syntax for a while loop in our language:

```
while condition
    what we want the program to do while the condition is true
endwhile
```

The condition in a while loop is similar to the one we learned in Chapter 3 for if statements – it is a Boolean expression, written in some quantifiable way – and we will apply the relational operators that we learned there, with loops as well.

The beginning of the loop – `while condition` – is referred to as the loop’s **header**, and the contents of the loop, between the header and `endwhile`, are referred to as the loop’s **body**. The `endwhile` keyword simply marks the end of the loop’s body: it is not an instruction that says to stop executing the loop – the loop stops executing only when its condition is no longer true. Each execution of the body of the loop is called an **iteration**.

In order to be able to write while loops, we have to learn a couple of preliminary topics.

Counters

A counter is simply a variable that is used to keep a running count of something. Let us refer back to our earlier example of signing students up for a class so long as the number enrolled in it is less than 24. For this, we could declare an integer variable named `studentsEnrolled`, in which we plan to keep our running count. In many cases, as in this one, the counter goes up in increments of 1 – we can see this logically by realizing that we usually sign up one student at a time for a class, rather than a group of students all at the same time. A different counter, say, for the number of tickets sold for a movie, could go up in larger increments, since a group of people may come up and purchase several tickets in one transaction. This brings us to the question of, how do we make any counter's value go up, or, in other words, how do we increment any counter whether it is in steps of 1 or more? Let us consider the `studentsEnrolled` counter – each time we sign up one more student, the counter's value has to increase by 1. We can think of this in two steps:

1. Take the current value of the counter and add 1 to it
2. Make that the new value of the counter

In code, this would look like:

```
studentsEnrolled = studentsEnrolled + 1
```

The critical part to be clear about here is that the `=` operator in programming is not making a statement that the items on the left hand side and right hand side of it are equal, but rather, it represents an instruction that says “take whatever is on the right hand side and make it the value of the variable shown on the left hand side”. So, if `studentsEnrolled` was already 5, then the line of code shown above would calculate $5 + 1$, which gives us 6, and store that 6 into the `studentsEnrolled` variable. Keeping in mind that variables identify locations in the computer's RAM, we would say that that RAM location contained 5, but now we are overwriting the 5 with 6.

Accumulators

An accumulator is a variable that is used to keep a running *total* of something. For an example of this, let us refer back to the scenario of a college wanting to calculate the total enrollment in all of its courses. We could declare an integer variable `totalEnrollment` in which we plan to do this calculation – this would be our accumulator. Likewise, we could declare another integer variable `courseEnrollment`, which would contain the enrollment for the individual course that we are considering at the moment. Thinking about the logic for this, we can see that we would be adding the `courseEnrollment` for each course to the `totalEnrollment`, giving us a running total of the enrollment in all of the courses we have considered so far. The line of code for this would look like:

```
totalEnrollment = totalEnrollment + currentEnrollment
```

As we did with the `studentsEnrolled` counter earlier, let us apply some numbers to see how this line of code would operate. Suppose `totalEnrollment` happens to be 60 at the moment. We access the next course's enrollment, which is, say, 17, and is stored in `currentEnrollment`. Our line of code above would add 60 and 17, making 77, and then assign that value back to `totalEnrollment`.

Loop structure and execution

Let us write some a simple `while` loop to start with – one that simply outputs the numbers 1 through 20, and understand its logic.

```
integer nextNumber
nextNumber = 1
while nextNumber <= 20
    output "The next number is: " + nextNumber
    nextNumber = nextNumber + 1
endwhile
```

Let us work through this example's logic, to practice using the terminology and also understand in detail the mechanics of a loop:

- a. before the loop, the variable `nextNumber` is initialized to the first of the numbers that we want to output, i.e., 1
- b. `nextNumber <= 20` is the loop's condition, and this loop will execute so long as that condition is true
- c. `while nextNumber <= 20` is the loop's header, and it shows when this loop will continue to execute – so long as `nextNumber <= 20`
- d. the two lines of code that output the next number and increment `nextNumber` by 1 are the body of the loop
- e. Each time we execute this loop's body, i.e., on each iteration of this loop, we output the current value of `nextNumber` and then increment that value by 1 – in this way, we use the `nextNumber` variable to generate each number that we want to output
- f. Before each iteration, we test the loop's condition, and do one more iteration only if the condition is still true
- g. When `nextNumber` has the value 20, we output that and increment it one last time, making its new value 21
- h. This time, we test the loop's condition, find that it is no longer true, and therefore, do not do any more iterations of the loop, and simply skip past `endwhile` to whatever other code our program may contain there.

Since the `nextNumber` variable controls the loop, i.e., its value determines whether the loop will have one more iteration, it is said to be the **loop control variable** in this case.

With the above details of a loop's structure and logic in mind, we can come up with a general recipe of sorts for writing a loop:

1. Declare the loop control variable (the one with which you plan to control the loop)
2. Initialize it to an appropriate value before the loop, e.g., in this case, we initialized `nextNumber` to 1, as that was the first of the numbers we wanted to output. Think about what would have happened if we had initialized it to, say, 30, instead – the loop would not have executed at all, since 30 is not ≤ 20 , making the loop's condition false from the start.
3. Use it in the loop's condition
4. Somewhere within the loop's body, modify the loop control variable in some way that fits the context, e.g., in this case, we wanted to use it to generate the next bigger integer than the one we just now output, *and will eventually make the loop's condition false*. For instance, in this example, if we had done

```
nextNumber = nextNumber - 1
```

we would not have generated the next number that we wanted to output, *and the loop would never have stopped at all as its condition would never have become false* – in other words, we would have written an **infinite loop**.

As we will see later, when we look at the types of loops, the same recipe applies no matter what type of loop we are writing, or what the loop's purpose is.

Try It:

Write similar simple `while` loops that do the following:

1. Output the even numbers only, from 2 through 500
 2. Output all numbers from 200 down to 50
 3. Output all multiples of 5, from 5 through 1000
-

Section 2: Types of Loops

Types of loops

There are two types of loops:

1. Counter-controlled loop
2. Sentinel-controlled loop

Counter-controlled loop

As the name suggests, this is the type of loop that is controlled by a counter. In the example that we just looked at, of a loop that outputs the integers 1 through 20, even though the `nextNumber` variable is not strictly a counter, i.e., it does not actually count anything, we could still consider this to be a counter-controlled loop. Before we go on to take a look at sentinel-controlled loops, let us see a complete program that contains a true counter-controlled loop, uses an accumulator as well, and consider the two possible variations of such a loop.

For this, let us write a program that does the task we mentioned as an example earlier – of calculating the total enrollment at a college. In this program, we will

1. Initially, assume a set number of courses, say, 200.
2. Allow the user to enter each course's enrollment
3. Accumulate the total enrollment for the entire college
4. Calculate the average course enrollment

Here is what that would look like:

```
startprogram EnrollmentAnalyzer
  startmain
    const integer COURSES_OFFERED = 200
    integer coursesProcessed
    integer courseEnrollment
    integer totalEnrollment
    floating averageEnrollment

    output "Welcome to the Enrollment Analyzer."
    output "This program will allow you to enter " +
      "course enrollments and will calculate " +
      "total and average enrollments."

    coursesProcessed = 0
```

```

totalEnrollment = 0

while coursesProcessed < COURSES_OFFERED
    output "Enter the enrollment in the next course: "
    input courseEnrollment
    totalEnrollment = totalEnrollment + courseEnrollment
    coursesProcessed = coursesProcessed + 1
endwhile

averageEnrollment = totalEnrollment / COURSES_OFFERED

output "The total enrollment: " + totalEnrollment
output "The average enrollment: " + averageEnrollment
output "Thank you for using the Enrollment Analyzer."

endmain
endprogram

```

In this program, we can see the following

1. `coursesProcessed` is actually a counter – it keeps a running count of how many courses' enrollments have been processed so far (i.e., entered and added to the running total)
2. `totalEnrollment` is an accumulator used to keep a running total of the course enrollments
3. We follow the general recipe for writing a loop outlined earlier, with one additional step before the loop – initialize the accumulator to zero.

Definite vs Indefinite loops

A **definite loop** is one in which we know the number of iterations it will have at the time we are writing it, and it will always have the same number of iterations. All of the example loops that we have seen so far are definite loops: the one that outputs integers 1 through 20 will always have 20 iterations, which we knew right when we wrote it; likewise, the loop in the `EnrollmentAnalyzer` program will always have 200 iterations, or more accurately, `COURSES_OFFERED` iterations, since we could open the program's code and type in a new value for that constant – if we were to that and change its value to 500, then we would know that the loop will always have 500 iterations.

In contrast, an **indefinite loop** is one in which we do not know how many iterations it will have when we are writing it. For an example of this, suppose we modify our EnrollmentAnalyzer program to allow the user to enter the number of courses offered by the college – in that case, we could not have a constant for that number, but would have a variable instead, whose value would be different each time we run the program, based on what the user enters for the number of courses offered. Here is a version of that program modified in that way:

```
startprogram EnrollmentAnalyzer
    startmain
        integer coursesOffered
        integer coursesProcessed
        integer courseEnrollment
        integer totalEnrollment
        floating averageEnrollment

        output "Welcome to the Enrollment Analyzer."
        output "This program will allow you to enter " +
            "course enrollments and will calculate " +
            "total and average enrollments."
        output "Enter the number of courses offered: "
        input coursesOffered

        coursesProcessed = 0
        totalEnrollment = 0

        while coursesProcessed < coursesOffered
            output "Enter the enrollment in the next course: "
            input courseEnrollment
            totalEnrollment = totalEnrollment + courseEnrollment
            coursesProcessed = coursesProcessed + 1
        endwhile
```

```

        averageEnrollment = totalEnrollment / coursesOffered

        output "The total enrollment: " + totalEnrollment
        output "The average enrollment: " + averageEnrollment
        output "Thank you for using the Enrollment Analyzer."

    endmain
endprogram

```

With this version, we cannot say at the time of writing the program, how many iterations the while loop in it will have, making it an indefinite loop.

Sentinel-controlled loop

This type of loop is controlled not by a counter, but by the user's entry or data from some other source, that equals some predetermined value or some event occurring. As an example of the latter, consider a program that reads data from a file – the program does not know where the end of the file is, but instead, a loop within the program would read data from the file *so long as we have not reached the end of the file*. You can see that the loop's condition in this case is more open-ended – continue to read from the file so long as we have not reached its end.

We could have the same sort of open-ended nature to the user's input, i.e., let them continue entering data and when they reach the end of their data, enter some specific value to signal to our program that they have finished. This specific value that we require the user to enter to indicate the end of their input is called a **sentinel**, hence the name for this type of loop – sentinel-controlled loop.

You can also see that this type of loop is by its very nature an indefinite loop – there is no way to know when we are writing the program, when the user will enter the sentinel and therefore, how many iterations our loop will have.

For an example of this type of loop, let us modify our EnrollmentAnalyzer program again – this time, we will not require the user to tell us at the beginning how many courses are offered; instead, we allow them to indefinitely enter course enrollments, and when they are done, enter a specific value to indicate that they are done.

This brings us to a discussion of how to choose a sentinel value. The two rules to follow here would be

1. The sentinel has to be of the same datatype as the normal input, e.g., since the course enrollments are integers, the sentinel has to be an integer as well. This is because the user would be entering the sentinel value *instead of* the next course's enrollment when prompted for that.

2. The sentinel's value has to be outside of the normal range of input, e.g., 20 would not be a good choice for a sentinel value in this program, since that is a valid enrollment value for a course; however, -1 would not be a valid enrollment value, and would therefore make a good sentinel.

Following the good programming practices shown in Chapter 1, declare a constant to represent the sentinel value and use that constant throughout the program's code in every place where it applies.

Here is a version of the program, modified accordingly. Note that in this case, `courseEnrollment` is the loop-control variable, and it is initialized before the loop by having the user input either the enrollment for the first course or the sentinel. This is called a **priming input**. This approach serves two logical purposes – initialize `courseEnrollment` before the loop as our recipe for writing loops tells us to, and *allow the user to exit immediately if they want to*. We do not want to write the sort of bossy program that illogically makes users do something at least once even when they don't want to, nor introduce errors into the program by having it do unnecessary operations when the user is trying to exit.

```
startprogram EnrollmentAnalyzer
```

```
    startmain
```

```
        const integer DONE = -1
```

```
        integer coursesOffered
```

```
        integer courseEnrollment
```

```
        integer totalEnrollment
```

```
        floating averageEnrollment
```

```
        output "Welcome to the Enrollment Analyzer."
```

```
        output "This program will allow you to enter " +
```

```
                "course enrollments and will calculate " +
```

```
                "total and average enrollments."
```

```
        output "Enter the enrollment in the first course " +
```

```
                (or enter " + DONE + " to stop): "
```

```
        input courseEnrollment
```

```
        coursesOffered = 0
```

```

totalEnrollment = 0

while courseEnrollment != DONE
    totalEnrollment = totalEnrollment + courseEnrollment
    coursesOffered = coursesOffered + 1
    output "Enter the enrollment in the next course " +
        (or enter " + DONE + " to stop): "
    input courseEnrollment
endwhile

averageEnrollment = totalEnrollment / coursesOffered

output "The total enrollment: " + totalEnrollment
output "The average enrollment: " + averageEnrollment
output "Thank you for using the Enrollment Analyzer."

endmain

endprogram

```

In this program, you can see that we still apply the same recipe for writing loops, but the lines of code that implement the recipe look different:

1. Declare the loop control variable (the one with which you plan to control the loop) – we declared `courseEnrollment` before the loop.
2. Initialize it to an appropriate value before the loop – we initialized it before the loop with our priming input, allowing the user to enter either an actual course's enrollment or the sentinel value of `-1`.
3. Use it in the loop's condition – the loop's condition is `courseEnrollment != DONE`, i.e., the loop should continue to execute so long as the user has not entered the sentinel.
4. Somewhere within the loop's body, modify the loop control variable in some way that fits the context, e.g., in this case, at the end of the loop, after we have added the current course's enrollment to the running total and incremented our count of courses offered, we ask the user to enter either the next course's enrollment, or the sentinel if they are done. In response to this request, at some point, the user will enter the sentinel value of `-1`, which will cause our loop to stop executing. If we fail to do this step, we will have written an infinite sentinel-controlled loop.

What to include in a loop

Now for a general rule to follow when writing loops, to make them easier to write, read and update in the future:

A loop is an iteration structure, i.e., one that repeats the same action several times. With this in mind, in each loop, include only the specific steps that need to be repeated. For instance, in the `EnrollmentAnalyzer` example shown earlier, the loop contains only the steps that need to be repeated for each course – add the course’s enrollment to the running total, increment the number of courses processed/offered, as appropriate for the different versions of that program. All the code for declaring the variables and constants, initializing the loop control variable, is all preliminary to the loop, and is written before it. The calculation of the average enrollment and output of the total and average enrollments needs to be done only one time, after we know what the total enrollment for all courses is, so, it comes after the loop.

An important addition to this, which we will see in later chapters, is what to include in the loop’s header. For the moment, we will state a general rule for that – any loop’s header must show *all of the conditions under which the loop should continue executing*. In other words, do not hide some conditions under which the loop should *stop* executing within the loop’s body – capture them all within the loop’s header.

Chapter 5

Modular Programming

Introduction to the Chapter

In Chapters 3 and 4, we learned the set of control structures that can be used to solve any programming problem that has a solution. Having learned those structures, viz., sequence, selection and iteration, we now have the tools to solve programming problems. In any professional programming language that you learn, you will encounter the same control structures. Each language may contain some variations of the structures that we learned, or additional types of selection structures or loops, but you can understand and use them all based on what we have learned here. Now, in Chapter 5, we turn our attention to code organization, i.e., the question of how we can better organize our code to make it easier to read, update, and maintain in the future.

Learning Outcomes

After reading this chapter, students will be able to:

1. Describe what a module is.
2. Describe what modular programming is.
3. Describe the benefits of modular programming
4. Write programs in AJANRO that utilize modules.

Section 1: What is Modular Programming?

In order to learn what modular programming is, we first have to learn what a module is.

We can define a **module** in programming simply as – a block of code that has a name and that performs one specific task needed by the program.

Modular programming would be the activity of organizing a program into a set of modules, each of which does a different task required by the program.

Before we take a look at how to write a module, and how to write a modular program, let us understand the benefits of this kind of code organization.

Most of the programs we have written so far have been small, perhaps a few dozen lines of code each. However, professional software products are large programs, containing thousands of lines of code. In such large programs, it would be extremely difficult to locate desired sections of code that need updating, if the entire program were a single large block of code in the way we have been writing programs so far.

Think of large programs as being similar to complex assemblies like any other common product with which you are familiar. Let us take a computer for instance – each computer contains several smaller complex parts, e.g., the hard disk assembly, the power supply, the motherboard, and so on. If you think of each of these complex parts as a separate module, then you can see that each such module can be manufactured and/or assembled separately and in parallel with the other modules, and then the modules assembled to make the complete machine. Likewise, large programs can consist of individual code modules that are “manufactured or assembled” in parallel by different programmers or teams of programmers, and then those modules are put together to build the complete program.

Programs often require the same task to be done, e.g., the same complex decision to be made, or the same multi-step calculation to be performed, at various points. Repeating the same code at each such point in the program creates two problems

1. Updating the code that performs that task requires us to locate all of the places within the program where we have repeated that code, and then perform the same update several times.
2. The likelihood of our missing some of those places is high, resulting in an inconsistent program.

You may ask, why is maintaining/updating a program easily such an important consideration? Think about all the programs that you use – some of them, you have been using for several years, while having to download updates when those are available. What this tells us is – the original creation of a program is a one-time activity, and is the smallest portion of the program’s lifecycle; any successful program then spends the rest of its lifecycle – years or decades – in being updated and maintained. It only make sense then that we should make the task involved in the major part of a program’s lifecycle as easy as possible.

Structure of a module

Now that we know why modular programming is beneficial, let us learn how to do it.

Every programming language supports modules in some way – most commonly, a language contains just one type or variety of module; occasionally, you may see a language that supports more than one type of module. As we have done with the control structures in our AJANRO pseudocode language, to keep things simple, we will learn just one type or variety of module. The terminology that is used for modules in different languages can also be different – you may see them referred to as functions, or methods, or procedures, or routines and so on. Some of those terms do mean somewhat different things, but overall, they all belong under the general category of modules. So as to keep things simple again, and more general, in our language, we will simply refer to them by the general term of “module”.

Further, let us look at one detail of how modular programs work, to help us understand the terms used in describing the structure of a module. Since modules are independent blocks of code that each perform one individual task that the program requires, on their own, they do not know *when that task is to be performed*. The job of orchestrating all of the modules, and getting them to do their task when needed, falls to the program's **main module** or **mainline logic**, as it is called. The part in which you have been writing all of your code in our language so far – between `startmain` and `endmain` – is actually the mainline logic. In other words, in each program so far, you have been writing only the program's main module, and it has been doing all necessary tasks on its own, with no helpers, but now we are about to give it some assistants whom it can call upon to do the various tasks required in the program. Whenever you write a line of code where you cause one of those modules to execute, you are said to be calling that module. Any module that calls another module in this way is said to be the caller. An important aspect to note from those last two sentences is that the mainline logic may not always be the caller, i.e., the mainline logic may call one module, but that module's task may be complex enough that it has to call another module to help it do part of its task.

Here, then, is the general syntax/structure for a module in our language:

```
return-type moduleName(parameter list)

    code to do the module's specific task, including declarations
    of any

    variables needed by this module to do its task, any calculations
    and

    so on

    return statement

endmodule
```

Let us look at what the individual parts are in this structure, and then we will apply them to write a small module that does one small, specific task.

return-type – this would specify what datatype of result this module will return to its caller upon completing its task. For instance, a module whose task is to calculate the retail price of an item would return a result of the floating datatype in our language. A different module, whose task is to make a decision of some sort, would return true or false, i.e., the `boolean` datatype, in our language. Think of the return-type as the module communicating in advance to anyone who might call it, saying, “if you call me, here is the type of result I will return to you, so, be prepared to receive that type of result”. So, what are the possible return-types for any module? Any datatype in our language is a valid return-type. We may have to write modules that simply perform some task, e.g., show a message, but do not have any result to return – in those cases, the module's return type would be `void`.

moduleName – this one is easy to describe – this would be a name that we choose for this module when we are writing it, which describes what task this module will perform. This is similar to the way we choose descriptive names for variables and constants, that explain what those represent within the program. In many languages, names of modules are typed in `camelCase`, just like names of variables are, and we will follow the same practice in our

language. So, our example module that calculates the retail price of an item, would be named `calculateRetailPrice`.

parameter list – this is a comma separated list of the pieces of data that this module requires in order to do its task. For instance, our `calculateRetailPrice` module would need to know two pieces of data, viz., what is the original cost of the item, and what is our percentage markup. Each such piece of required data is called a formal parameter or simply parameter. Each parameter has a name that describes what it represents, and a data-type. Again, this is a way for the module to communicate, “if you want me to do my task, here are the pieces of data that I need, and here are their datatypes”.

return statement – this is a single line of code where the module actually returns a result of the data-type that it has specified in its return-type earlier; in the case where the module’s return-type is void, we would simply write an empty return statement that contains just the return keyword and nothing else. In either case, the return statement has two purposes – return the result that the module has generated, if any, and *return control to the caller* so that they can go on to the next step within the program’s overall sequence.

`endmodule` – similar to `endprogram`, `endmain`, `endif` and `endwhile`, which we learned earlier, this is simply a code marker that says where this particular module’s code ends.

A couple of further pieces of terminology associated with modules – the entire first line shown above, that contains the return-type, module’s name and parameter list, is said to be the module’s **header**, and the code contained within the module is said to be the module’s **body**.

Writing a simple module

Now that we have learned the structure and general syntax of a module in our language, let us try writing a simple one. At the moment, we will not try to write the rest of the program, but rather, simply write a module that can perform one simple task when it is called, and return a result. To do this, let us write the `calculateRetailPrice` module that we have been using to describe the structure of a module:

```
floating calculateRetailPrice(floating cost, floating
markupPercent)

    floating retailPrice
    retailPrice = cost + (cost * markupPercent / 100)
    return retailPrice
endmodule
```

Let us apply the terminology that we learned earlier, to this example module, and identify its parts.

The module’s header is

```
floating calculateRetailPrice(floating cost, floating
markupPercent)
```

Within the header, the various parts are

return-type – `floating`

module's name – `calculateRetailPrice`

parameter list – `floating cost, floating markupPercent`

Within the parameter list, we have two parameters, named `cost` and `markupPercent` respectively, and they are both of the `floating` datatype. (All parameters of a module need not be of the same data-type – they just happen to be so in this example.)

Let us describe in words what this header is trying to communicate to anyone who plans to call this module – it says: “if you call me and give me the cost of an item and the markup percentage that you want applied to it, both of the `floating` data-type, then I will calculate the retail price, just as my name indicates, and will return that retail price to you – it will be of the `floating` data-type, so, be prepared to receive a `floating` result from me.”

Before we move on to describing the remaining parts of the module, let us pause for a moment and think about why the module's return type is `floating`, and so are the data-types of its parameters – these are simply logical choices of the sort we have been making since Chapter 1, i.e., we know logically that an item's cost is of the `floating` data-type, since it would be, for instance, \$3.57; likewise, we logically know that a markup percentage and retail price would also be of the `floating` data-type. In other words, we simply apply the same logical decision-making process that we have been applying all along to determine datatypes for variables and constants, within this new context of writing a module.

Now, let us continue identifying the remaining parts of this module.

The module's body is all of the lines of code below the header and above the `endmodule` marker.

Within the body, we have a declaration of a variable named `retailPrice`, a line of code that calculates the retail price based on the information that is given to this module, i.e., the `cost` and `markupPercent`, and finally, the return statement.

Section 2: Benefits of Writing a modular program

Writing a modular program can be divided into two parts

1. Design the modules needed by this program and then write them – this means, figure out what each module's header will be, what its body should do, and then write each module.
2. Write the program's mainline logic, which will call the modules when the program logically needs each module's task to be performed.

So far, we have tried the first step, and written a single module for a program whose job is to calculate the retail price of an item and then output that.

Now, let us try out the second step. For this purpose, let us assume that the program's overall responsibilities are as follows:

1. Get the cost and markup percentage as input from the user
2. Calculate the retail price of the item
3. Output the retail price

In order to accomplish these, the mainline logic has to get the necessary input first, call the module that does the calculation of the retail price second, and then output that retail price third.

Here, then, is the complete program that does this, including the module that we wrote earlier and the mainline logic:

```
startprogram RetailPriceCalculator

    floating calculateRetailPrice(floating cost, floating
markupPercent)

        floating retailPrice
        retailPrice = cost + (cost * markupPercent / 100)
        return retailPrice
    endmodule

startmain

    floating cost
    floating markupPercentage
    floating retailPrice

    output "Welcome to the Retail Price Calculator."
    output "This program will allow you to enter " +
        "an item's cost and markup percentage " +
        "and will calculate its retail price."
    output "Enter the item's original cost: "
    input cost
    output "Markup percentage (e.g., for 5%, enter simply 5:
"
    input markupPercentage
```

```

        retailPrice = calculateRetailPrice(cost, markupPercentage)

        output "The retail price of the item is: $" + retailPrice
        output "Thank you for using the Retail Price Calculator."
    endmain
endprogram

```

You are familiar with most of what the mainline logic, i.e., the part between `startmain` and `endmain`, contains, i.e., declarations, lines of code that prompt for and get the user's input, and code that shows output.

The part that is new here is the one line of code that calls the `calculateRetailPrice` module –

```
retailPrice = calculateRetailPrice(cost, markupPercentage)
```

Let us analyze this line of code.

The `calculateRetailPrice` module has let us know what data the module requires, and what we should be prepared to receive from it when we call it. So, we prepared accordingly, i.e., we

1. Got the user's input of the item's cost and markup percentage so that we could give those, or more technically, *pass* those, to the module.
2. Declared a variable named `retailPrice`, in which we could store whatever the module returns to us.

Making use of these, then, we called the `calculateRetailPrice` module and passed it the cost and markup percentage that were input by the user. However, we knew that it was going to return the retail price that it calculated, which was going to be of the floating data-type, so, we took what the module returned and assigned it to our `retailPrice` variable, so that we could later go on and output that retail price.

From this, we can see in general how to call any module that returns a result:

1. prepare in advance by declaring a variable that has the same data-type as the module's stated return-type
2. call the module on the right-hand side of an assignment statement
3. assign whatever the module returns to the variable we declared for that purpose in step 1 above

In order to contrast this with calling a module that *does not return a result*, i.e., a module whose return-type is `void`, let us revise our program by adding another module whose task is

simply to output the retail price and do nothing else. In the program below, you will see a `showRetailPrice` module below the one we had already written. Note how, since its return type is `void`, its return statement is empty, i.e., it simply returns control to whoever called it, but does not return any information/result to them.

```
startprogram RetailPriceCalculator

    floating    calculateRetailPrice(floating    cost,    floating
markupPercent)

        floating retailPrice

        retailPrice = cost + (cost * markupPercent / 100)

        return retailPrice

endmodule

void showRetailPrice(floating retailPrice)

    output "The retail price of the item is: $" + retailPrice

    return

endmodule

startmain

    floating cost

    floating markupPercentage

    floating retailPrice

    output "Welcome to the Retail Price Calculator."

    output "This program will allow you to enter " +
        "an item's cost and markup percentage " +
        "and will calculate its retail price."

    output "Enter the item's original cost: "

    input cost

    output "Markup percentage (e.g., for 5%, enter simply 5:
"

    input markupPercentage
```

```

        retailPrice = calculateRetailPrice(cost, markupPercentage)

        showRetailPrice(retailPrice)

        output "Thank you for using the Retail Price Calculator."
    endmain
endprogram

```

Looking at the line of code above that calls the `showRetailPrice` module, viz.,

```
showRetailPrice(retailPrice)
```

we can see how this is different from the previous line that calls the `calculateRetailPrice` module. In this case, since we expect no result back from the `showRetailPrice` module, we simply call it by name and pass it the data it needs from us.

Parameters and arguments

From the example program that we have written above, let us note a couple of subtle points.

1. When writing the `calculateRetailPrice` module, we did not actually have any data yet with which to calculate a retail price. Indeed, the module's header states, "if you give me these two pieces of data, I will calculate the retail price and return that to you." However, it does assign names to the two pieces of data that it expects, so that its code could use those names to refer to those pieces of data in its calculations. This, then, would be what a module's **parameter** is – simply a name by which the module refers to data that it expects to receive. In the case of the `calculateRetailPrice` module, those names are the ones it includes in its parameter list – `cost` and `markupPercent`.
2. At what point does the module have any actual data to work with? When it is called and passed some data! In our program, this happens when the mainline logic calls the `calculateRetailPrice` module and passes it the contents of its `cost` and `markupPercentage` variables, which were obtained as input from the user. The actual data that we pass to a module when we call it are said to be **arguments**.

You may see these terms – parameter and argument – used interchangeably in textbooks and other programming guides.

Flow of control

Now that we have seen how to write a modular program, let us think about how the program would execute. For this, let us again use the `RetailPriceCalculator` program.

In any program in any language, execution begins with the mainline logic. In other words, when the program starts running, it will start by doing whatever the code within the mainline logic says.

So, in our program, the mainline logic starts out by declaring the three variables for cost, markup percentage and retail price. It then goes on to show a welcome message. It then prompts the user to input the cost and markup percentage.

The next line of code, however, calls the `calculateRetailPrice` module. At that point, we leave the mainline logic, go into that module, and execute its code. That module's code declares a variable for the retail price that it is going to calculate, and performs that calculation. When the next line of code within it – the return statement – is executed, at that moment, we leave that module and return to the mainline logic.

At that point, whatever value the module return is now assigned to the `retailPrice` variable that was declared within the mainline logic, and then we continue with the mainline logic's code.

The next line of its code calls the `showRetailPrice` module, so, we now enter that module, execute its code, and then its return statement causes us to return to the mainline logic.

Finally, we execute the remaining lines of code within the mainline logic, and when we reach the `endmain` marker, we are done, and our program exits.

Modular program design

In our example program, you can see that we first wrote just one module – the `calculateRetailPrice` module, but then we wrote an additional module to show the retail price.

This brings us to a discussion of modular program design – in other words, how to decide what modules to write. A general rule to follow is, if there is any multi-step task that the program's logic requires, extract that into a separate module. This is in anticipation of the possibility that

1. the steps involved in that multi-step process may change or become more complex in the future, in which case, it would be easier to just locate that module within the program and apply those changes
2. the program may need that same task performed at more points in its logic in the future, in which case, at all those points, all we would have to do is insert single lines of code that call the module that performs that task

By planning ahead for what may come in the program's future and writing modules within it accordingly, we make the task of maintaining or updating the program easier.

So, a simple process you could follow in writing modular programs is:

1. Make a list of the individual logical tasks that the program has to perform in order to fulfil its overall responsibilities.

2. Write modules that perform each of those tasks.
3. Write the mainline logic such that it calls the modules in the correct sequence within its code, to complete what the program needs to.

Section 3: Scope of Variables

In the example programs we have seen so far in this chapter, you will notice that we have variables with identical names in both the mainline logic and some of the modules. Do these names not conflict with each other, then? No!

Each variable that we declare within any block of code exists only within that block – a variable declared within a module exists only within that module; a variable declared within the mainline logic exists only within the mainline logic. This leaves us free to reuse names that refer to the same logical quantity within various modules in our program. The same applies to constants as well, so, as you read the rest of this section, understand that most of what is described here with reference to variables applies to constants too.

The block within which any variable exists, or is available/accessible to be used, is called its scope. For example, we declared a `retailPrice` variable within the `calculateRetailPrice` module in our example earlier – the scope of that variable is just that module. What this means is, once we are outside of that module’s code, that variable ceases to exist, and we cannot refer to it anywhere outside of that module. Now, the mainline logic also declares a variable with the same name, `retailPrice` – any code within the mainline logic that uses that name is referring to that variable, and not to the one we declared within the `calculateRetailPrice` module that simply happens to have the same name.

So, if variables declared within the mainline logic are only available within it, what if the data they contain is needed by some other module that we plan to call, indeed, such as the `calculateRetailPrice` module? This is where module’s parameters come in – they are the means by which we can communicate data from the mainline logic to the modules it calls, and indeed from one module to another module that it happens to call.

Technically speaking, space in the computer’s RAM is allotted for variables declared within any module at the moment that module’s code is executed, as described in the section on flow of control earlier, and once we reach the end of that module, those spots in RAM are no longer considered necessary for our program, since that module may not be called again, and so, they are no longer available to our program. If our program’s code does call the same module again, then a fresh set of spots in RAM will be allotted for the variables that the module declares, again, available only so long as we are executing that module’s code.

Any variables that we declare within any module are said to be **local** to that module, and they are said to have **local scope**.

More accurately, any variables declared *within any block of code* are local to that block. For instance, many languages support types of loops where you can declare variables within the loop’s structure – those variables would be local to that loop only and would not be available outside of it.

This might make you wonder – can we declare variables that are not local? Yes. We can declare variables at the program level, i.e., outside of all modules and outside of the mainline logic. These would be declared immediately after the beginning of the program, i.e., just below the line that begins with `startprogram`. These variables are said to have **program-level scope**.

We will see examples of this in a later chapter, and there, will also learn what criteria to apply to decide when to declare local variables versus program-level ones. For the moment, we will learn a simple rule of programming – **always prefer local variables over program-level ones**. Another way of saying this is, declare program-level variables only when you absolutely have to, which, at the moment, we will not encounter any such situation until the next chapter.

The simple reason for this restriction is that *all* modules within a program can access any program-level variables, which means they can all assign new values to those variables, quickly making it hard to predict what values those variables might contain, and when they have incorrect or invalid values that cause our program not to function correctly, make it difficult to track down which module was the cause of that problem.

Program-level constants, however, do not suffer from that problem, because their values cannot be changed after they are declared. So, you are free to declare program-level constants and will see some of those in the last example in this chapter below.

Chapter 6

Arrays

Introduction to the Chapter

From Chapters 1 through 5, we learned about the control structures that are the building blocks of any program, and modular programming, which helps us organize our code better. Chapter 6 turns us in a different direction by introducing the concept of data structures, and then presenting in detail the simplest of the data structures – an array.

Learning Outcomes

After reading this chapter, students will be able to:

1. Describe what an array is.
2. Describe the basic array algorithms.
3. Describe/Explain the benefit of using arrays.
4. Write programs in AJANRO that utilize arrays.

Section 1: Understanding arrays

What is a data structure?

The introduction to this chapter mentions that an array is the simplest of the data structures. Let us first learn in general terms what a data structure is.

Chapters 1 through 4 presented the control structures in programming – these were structures that help us control the flow of a program, e.g., the program needs to choose one out of two or more alternatives, or, in terms of code, decide which one of two or more code branches to execute next, so, we write an if statement; the program needs to repeatedly do some task, so, we write a loop.

A **data structure** is one that stores a collection of *data* that the program needs, in the computer's RAM.

Recall from Chapter 1, that a program is a set of instructions that we write, which are being executed by the computer's CPU. We learned that programs store the data that they need to operate on in RAM. We learned how to declare variables, which are simply locations in RAM that the program plans to use, and so we assign names to those locations.

So far, our programs have only operated on single pieces of data at a time, e.g., our EnrollmentAnalyzer example program from Chapter 4 considered one course's enrollment at a time – at no point did it have the enrollments of *all* of our courses in RAM at the same time. A more likely scenario is one in which a program requires a *collection of data* to be present in RAM.

Let us consider a few such cases:

1. The EnrollmentAnalyzer program needs to show a sorted list of all our course enrollments, from lowest to highest.
2. We want to enhance the same program by providing a search function – the user inputs a course's title or course number, and our program will search for that course and output what its current enrollment is.

Neither one of these is an operation that could be done without having data about all of the courses in RAM at the same time. In order to sort the enrollments as described in #1 above, the program needs to have all of the course enrollments at the same time so that it can compare them with each other and figure out which ones are lower/smaller and which ones are higher/larger. In order to search for a desired course title, the program needs to have all of the course titles at the same time so that it can look for a match with the desired title. In other words, our program would need to store a collection of course enrollments for #1 above, and in the case of #2, would need to store *two* collections – one containing all the course titles and another containing the enrollments in those courses.

A further detail that we can gather from the examples above is, when we say that a data structure is a collection of data, that entire collection of data has to have the same logical meaning, e.g., a collection of course titles or a collection of course enrollments. In other words, we could not, for example, have one collection that mixes both course titles and course enrollments – a simple reason we can list at the moment would be, for instance when we are trying to search for a desired course title, we have to be able to compare it with only all of the course titles to see if we have a match.

Now that we know what a data structure is – some kind of structure in the computer's RAM that allows us to store a collection of data that will be used by our program – let us learn about the simplest such structure, the array.

What is an array?

An array is a data structure that:

1. Has a fixed size
2. Stores the data in contiguous locations in the computer's RAM, i.e., in locations that are right next to each other.

Every programming language supports arrays, and they are often the building blocks for other more complex data structures.

As with variables, which we learned about in Chapter 2 and have used since then, an array has to be declared by the program before it can be used.

Declaring an array

In Chapter 2, we learned the syntax for declaring individual variables, which we have been using in the subsequent chapters:

```
data-type variableName
```

The syntax for declaring an array is mostly identical to this, with one additional detail – we are now trying to declare something that, as described in the previous section, is a fixed-size collection of data, so, we have to say what the size of the collection is as well. As in most programming languages, we will indicate this inside a pair of square brackets:

```
data-type arrayName[size of collection]
```

So, for example, if we had 10 courses for which we wanted to store enrollments, we would declare an array of the integer data-type:

```
integer enrollment[10]
```

Even though an array is a collection of data, the practice is for the array's name itself to be a singular noun, such as `enrollment`, `firstName`, `title`, and so on. We will see why shortly. At the moment, let us fully understand both our declaration and the fact that an array is a contiguous structure. Upon executing the declaration above, in RAM, we end up with 10 locations right next to each other as shown below.

Table 6.1 Picture of an array in RAM

Each of these 10 locations is capable of storing one integer. The entire set of 10 locations has the same name – `enrollment`.

Try It:

Declare arrays with appropriate data-types and sizes to represent each of the following collections of data:

<come up with at least five different example collections for students to try>

Before we go on to the next topic, let us learn a piece of terminology associated with arrays.

In the above example, we declared an array named `enrollment`, of size 10, i.e., that has 10 contiguous locations in RAM, each of which can store the enrollment for one course. Each such location within an array is called an **array element** or simply, **element**. So, we would say that our `enrollment` array has 10 elements.

Storing data in an array

Now we know how to declare an array. How do we go about putting data into one? Again, let us start by looking at the syntax for storing data in a simple variable. The syntax we know for assigning a value to a variable ourselves within the program's code looks like this

```
variableName = value
```

and that for storing the user's input in a variable looks like this

```
input variableName
```

Again, the syntax for doing the same operations with an array is mostly identical to this, with one additional detail – the array's name refers to a collection of RAM locations rather than just one, or, to use the array terminology that we learned a moment ago, the array's name refers to the entire set of elements in that array. How, then, do we refer to just a single array element, to say something like, "I want to store this piece of data in the first spot in the array, i.e., in the first array element, or, "I want to store this data in the *second* array element" and so on? To see this, let us look at a revised picture of our `enrollment` array in RAM, and learn a second piece of array terminology:

Table 6.2 Picture of an array in RAM with indexes

0	
1	
2	
3	
4	
5	
6	

7	
8	
9	

Each array element is identified by a number – the first element is identified by the 0 that you see to its left in the figure, the second element by 1, the third one by 2, and so on. Each such identifying number is called an **array index**, or simply **index**. It is also referred to as a **subscript**.

We would say that our enrollment array has 10 elements and they are identified by indexes 0 through 9.

To access any element within an array for any purpose – storing data in that element, or using the data that is in that element in some operation – we would use the name of the array followed by the index inside a pair of square brackets like this:

```
arrayName[index]
```

So, to assign a value, or, in other words, to store some data in some specific array element, our general syntax would be

```
arrayName[index] = value
```

and to store the user’s input in some array element, the general syntax would be

```
input arrayName[index]
```

So, for example, if we wanted to store the value 15 in the first element of our enrollment array, our code would look like:

```
enrollment[0] = 15
```

To read or describe this line of code, you would say you were assigning the value 15 to enrollment at index 0, or to enrollment subscript 0.

On the other hand, if we wanted to, say, store the user’s input of a course’s enrollment in the second element of our enrollment array, our code would be

```
input enrollment[1]
```

Now we can see why the array’s name is usually a singular noun – our code usually operates on individual array elements by using the array’s name and an index, not usually on the entire array. Since we are referring to one single array element or one single piece of data within the array, it makes sense that the name by which we refer to it should also be singular – “enrollment at index 0” rather than “enrollments at index 0”.

Since the indexes begin at zero rather than 1, we would say that arrays use a zero-based index. While there is a more technical reason for this, for the moment, we can understand the zero-based index as telling us how many array elements we have to move past, starting from the beginning of the array, to reach the element that we want to access. So, to access the very first element, we start at the beginning of the array and advance past zero elements to reach the first element, we advance past 1 element to reach the second one, and so on.

Note that array indexes are not actually stored anywhere – in Figure 6-2, they are shown in the left column just to help us see how they identify each array element, however, the only place where they actually exist is in our code, to indicate, as described in the previous paragraph, how many array elements past the beginning of the array we need to advance in order to reach the one we want to access.

Try It:

Write lines of code to store data in the arrays as described in each example below:

<come up with at least five different examples using the arrays from the previous Try It exercise>

Array size and highest index

From Figure 6-2, which shows array indexes, you may have observed that with our array of size 10, the highest index turned out to be 9, or, in other words, the highest index was 1 less than the size of the array.

This relationship holds in any array –

Highest index = size of the array – 1

As you will see when we learn the array algorithms in detail, we can use this relationship to always be able to calculate what the highest index is in any array if we know the array's size.

Try It:

Calculate the highest index in the arrays described in each example below:

<come up with at least five different examples using the arrays from the previous Try It exercise>

Section 2: Array algorithms – The Power and Benefits

Using loops with arrays

The true power of arrays comes from our ability to use loops in conjunction with them. It is this feature that allows us to implement all of the array algorithms by which we can perform such operations as searching and sorting.

Using a variable as an array index

The ability to use loops with arrays depends upon the fact that we can use a variable as an array index. Let us see simply a demonstration of what this means, and then we will try writing a loop that fills our enrollment array with the user's input.

Suppose we declare an integer variable named `courseNumber`, to represent the number of the course on which we want to perform some simple operation, say, prompting the user to type in the enrollment for the first course and storing that input in our enrollment array. We could do this as shown below:

```
integer courseNumber
courseNumber = 0
output "Enter the enrollment for the first course: "
input enrollment[courseNumber]
```

This example code shows simply what using the `courseNumber` variable as an array index would look like. In the second line of code, we initialized `courseNumber` to 0, because we know that the very first array index is 0, and we want the user's input to be stored in the enrollment array at that index. Later, when the fourth line of code above is executed, the current value of `courseNumber`, i.e., 0, will be applied inside the square brackets, and the user's input will be stored successfully at index 0 in the enrollment array.

Let us try the same with a different index, say, 3. This time, let us revise our prompt for input – the user is no longer being asked to enter the enrollment for the first course. However, which course's enrollment would be located at index 3 in the array? Since array indexes begin at 0 rather than 1, we can figure out that index 3 refers to the *fourth* array element, and therefore, we are looking for the user to enter the enrollment for the fourth course. Right away, we notice that the course number is off by one from the array index, i.e., as far as the user is concerned, the first course is course #1, but as far as array indexes are concerned, that pertains to index 0; likewise, the fourth course is course #4, but its array index would be 3. Let us account for this difference by showing the user a prompt that contains a course number that would make sense to them, i.e., that the fourth course would be course #4 according to them, but their input would be stored at index 3 in our enrollment array.

```
courseNumber = 3
```

```
output "Enter the enrollment for course #" + (courseNumber + 1) + ":
"
```

```
input enrollment[courseNumber]
```

From the code above, we can observe two things:

1. Whenever we describe the contents of an array to the user – prompting them for input that will be stored in the array, or, as we will see later, displaying data that is already in the array – we have to adjust for the fact that arrays use a zero-based index, whereas users, i.e., people, begin counting at 1 rather than 0.
2. We are able to access *any array element at all at random* – we do not have to start at index 0 and work our way toward later array elements; instead, we can simply apply whatever array index we want to inside the square brackets, in order to refer to any desired array element. This feature will be useful in some of the later array algorithms that we will learn.

Having seen how we can use a variable as an array index, and how we can generate prompts that ask the user to enter the enrollment for any course number that we want to, let us now try writing a loop that will fill the entire enrollment array with the user's input. While writing this loop, we will apply what we learned about the highest index in an array: highest index = array size – 1.

```
integer courseNumber
courseNumber = 0
while courseNumber < 10
    output "Enter enrollment for course #" + (courseNumber + 1) +
": "
    input enrollment[courseNumber]
    courseNumber = courseNumber + 1
endwhile
```

As you can see, all we had to do was write a loop that is controlled by `courseNumber`, which is the variable we planned to use as an array index; the loop will continue so long as `courseNumber`'s value has not reached the array's size (we could have also written the Boolean expression as `courseNumber <= 9`, and it would have meant the same thing); on each iteration of the loop, we increment `courseNumber` by 1, so that we keep advancing through the array, storing the user's input at the next array index each time.

What if we had a larger number of courses, say, 1000, and so, we had declared our `enrollment` array with size 1000? All we have to do is replace 10 with 1000 in our loop's condition above, and the same loop would now fill an array of size 1000 with the user's input! From this, we can see how using loops with arrays is a powerful tool in programming.

Now that we know how to use loops with arrays, we are ready to start learning the array algorithms that we need to. Here, then, is a list of array algorithms that we will learn:

1. Filling an array – this one, we just learned above, where we wrote a loop that fills an array with the user’s input. The same principle would apply if we were trying to fill an array with data from some other source, e.g., from a file. We will revisit this algorithm later, when we learn how to *partially fill* an array, and work with partially filled arrays.
2. Outputting contents of an array
3. Calculating the sum (and average, which we will do later) of the elements of a numeric array
4. Scanning an array for maximum (highest) and minimum (lowest) elements.
5. Working with parallel arrays
6. Searching an array
7. Sorting an array
8. Working with partially-filled arrays

Since each array algorithm listed above performs a different task related to arrays, a study of them, and indeed, using them in practice later, lends itself well to modular programming. So, we will see each array algorithm applied in a separate module, all of which belong in one program, whose overall goal is to help us manage our courses. After we implement all of the array algorithms that we are going to learn within separate modules, we will write the mainline logic that declares the arrays that we need, and calls the individual modules when needed. To follow this approach, however, we will have to make a brief detour to see how to specify that a module has an array as a parameter.

Modules with array parameters

To indicate that a module has an array parameter, we simply place a pair of square brackets after the parameter’s name when writing the parameter list. After that, we are free to write code within that module that applies an array index to that parameter. Recall what a parameter is – data that a module requires in order to do its task – in this case, an array parameter would be a data *structure* that a module requires in order to do its task. In the next subsection, we will look at the second algorithm in our list – the one we would use to output all the contents of an array – at the moment, let us try just writing the header for a module that would accomplish the task of outputting all of our course enrollments.

```
void showEnrollments(integer enrollment[], integer coursesOffered)
```

With this header, this module communicates the fact that if it is given an array containing enrollment data, and given a count of the courses we offer, it will then show, or output, our enrollments for us. The square brackets after the parameter named `enrollment` indicate that this module’s first parameter is an array that contains enrollment data; you will notice that the second parameter – `coursesOffered` – does not have a pair of square brackets after it, indicating that is simply a single integer parameter, not an array, and its name shows that this parameter represents the number of courses we offer.

Let us now look at the algorithm for outputting all of the contents of an array.

Array algorithm 2: Outputting contents of an array

Both this algorithm and the one for filling an array can be summarized with the same four steps:

1. Start at index 0
2. Output whatever is at the current index in the array (or in the case of filling the array, store data at the current index in the array)
3. Increment the index on each loop iteration
4. Stop when we pass the highest index in the array

Writing the complete module for which we wrote a header earlier, it would look like:

```
void showEnrollments(integer enrollment[], integer coursesOffered)
    integer courseNumber
    courseNumber = 0
    output "Here are the current enrollments in all courses."
    while courseNumber < coursesOffered
        output "Course #" + (courseNumber + 1) + ": " +
            enrollment[courseNumber]
        courseNumber = courseNumber + 1
    endwhile
    return
endmodule
```

As you can see, the only difference between this algorithm and the one that fills the array is that this one shows output whereas the other one stores user input in the array. The loop itself is identical in both. In fact, it will be the same loop, with some modifications, that will appear in every array algorithm.

One other detail to observe is that the loop condition here uses the `coursesOffered` parameter, rather than a hard-coded 10. This makes the module more flexible – pass it any number in the second position, and it will think of that as the number of courses we offer. When we write a revised version of the loop that fills the array with the user's input, we will do something similar, so that the module itself makes no assumptions about exactly how many courses are being offered – instead, it allows the mainline logic (or any other module that calls it) to provide an argument for the actual number of courses offered, and it will then use that number in the loop's condition.

Array algorithm 3: Calculating the sum of the elements of a numeric array

This algorithm uses the same loop again, but requires that we declare an accumulator and initialize it before the loop, and then add the element at the current index to the running sum in the accumulator. The steps then, could be summarized as:

1. Declare an accumulator, initialize it to 0
2. Start at index 0 of the array
3. Add whatever is at the current index in the array to the accumulator
4. Increment the index on each loop iteration
5. Stop when we pass the highest index in the array

Writing the complete module for which we wrote a header earlier, it would look like:

```
integer sumEnrollments(integer enrollment[], integer coursesOffered)
    integer courseNumber
    integer totalEnrollment
    courseNumber = 0
    totalEnrollment = 0
    while courseNumber < coursesOffered
        totalEnrollment = totalEnrollment +
enrollment[courseNumber]
        courseNumber = courseNumber + 1
    endwhile
    return totalEnrollment
endmodule
```

Note that this time, the module actually returns an integer, which would be the total of all the course enrollments. Further, writing this module to calculate and return only the sum of the enrollment array's elements makes it more flexible as to how it can be used – the mainline logic may only want to get the total enrollment and output that, or it may do that and then use the total that was returned to also calculate the average.

Array algorithm 4: Scanning an array for maximum (highest)

In this subsection, we will see in detail how to scan for maximum, and then you will get to try modifying the code to scan for minimum (lowest).

In terms of our example program's logic, we would be trying to find out what the highest enrollment among all our courses is, and which course has that enrollment.

This algorithm uses the same loop again, but with a different couple of preliminary steps, as summarized below:

1. Assume that the very first array element – the one at index 0 – is the maximum. This is accomplished by declaring a variable to hold the maximum value, and initially assigning whatever is at index 0 in the array to it. Also, to keep track of *where* the current maximum is, we declare a separate variable to hold just the index of the maximum value, and initialize that to 0, following along with our initial assumption that the maximum value is at index 0.
2. Start at index 1 (rather than 0) of the array
3. Compare whatever is at the current index with the current maximum; if it turns out to be greater than the current maximum, then it becomes the new current maximum. Logically, we can understand this step as “we don't know if this is actually the maximum, but it is the maximum of the array elements we have examined so far”. Also, set the index of maximum variable to this current index.
4. Increment the index on each loop iteration
5. Stop when we pass the highest index in the array

When we exit the loop, whatever the maximum variable contains is the actual maximum, and the index of maximum variable will contain its index.

Writing the complete module for which we wrote a header earlier, it would look like:

```
integer    findHighestEnrollment(integer    enrollment[],    integer
coursesOffered)

    integer courseNumber
    integer highestEnrollment
    integer indexOfHighest
    highestEnrollment = enrollment[0]
    indexOfHighest = 0
    courseNumber = 1
    while courseNumber < coursesOffered
        if enrollment[courseNumber] > highestEnrollment
            highestEnrollment = enrollment[courseNumber]
            indexOfHighest = courseNumber
        endif
```

```

        courseNumber = courseNumber + 1
    endwhile
    return indexOfHighest
endmodule

```

Note that this time, the module returns the index of the course that has the highest enrollment. Again, this makes it more flexible as to how it can be used – the mainline logic may simply want to get and output which course number has the highest enrollment, or it may do that and then use the index that was returned to also access and output what the highest enrollment actually is as well.

Try It:

have students try to write a `findLowestEnrollment` module similar to this one. **MAYBE DEMONSTRATE CALLING THE FIRST MODULE AND THEN INCLUDE TRY ITS AT THE END OF EACH SUBSEQUENT SUBSECTION WHERE STUDENTS TRY TO WRITE A LINE OF CODE THAT CALLS THE MODULE.**

Array algorithm 5: Working with parallel arrays

This is not actually an algorithm, but rather, a programming technique that can be applied when we have more than one details about the same collection of items to maintain in RAM for the program to use. To apply this to our example of courses and enrollments, suppose we want to have not only the enrollments in all of the courses as we have done so far, but also have the titles of the courses, e.g., “College Algebra”, “Intro to Computers” and so on, so that we could allow the user to search for a course title: in that case, we would have two collections to store in RAM – a collection of course titles and a collection of course enrollments. How do we go about associating the titles with the enrollments in that case? Simply based on index, i.e., if we put a course’s title at index 0 in the course titles array, then we make sure to put the same course’s enrollment at index 0 in the enrollment array. Such arrays, that contain different details about the same item, where the details are kept associated with each other via array index, are called **parallel arrays**. What if we had a third detail, such as a course number, e.g., “MAT 101”, “CSC 101” and so on? We would declare a third array to store the course numbers, and again, maintain a parallel relationship with the other two arrays.

To see how to do this, let us go back and revise our code that declared the enrollment array, as well as our loop that stores the user’s input in the array. Since we have more than one array to declare, all of the same size, and that size could change in the future, we will make our program easier to update by declaring a constant that represents the maximum number of courses that we could offer, and then using that as the size of each array when we declare those, and use it in the loop’s condition as well. Also, since we have to prompt the user to input both the course title and the enrollment, we will try to simplify our prompts as you will see in the code below.

```

const integer MAXIMUM_COURSES = 10
string title[MAXIMUM_COURSES]
integer enrollment[MAXIMUM_COURSES]
integer courseNumber
courseNumber = 0
output "Please enter the details for each course when prompted."
while courseNumber < MAXIMUM_COURSES
    output "Course #" + (courseNumber + 1)
    output "Title: "
    input title[courseNumber]
    output "Enrollment: "
    input enrollment[courseNumber]
    courseNumber = courseNumber + 1
endwhile

```

As you can see from the code above, establishing a parallel relationship between two or more arrays is easy to do – in the loop above, we simply used the same array index on both the title and enrollment arrays when storing data in them, so that course titles and enrollments ended up at identical indexes in the two arrays. The modules that we have written so far, that utilize the array algorithms that we have learned so far, are easy to modify where needed, to take advantage of the fact that we now have a parallel array containing course titles.

- The showEnrollment module can easily be modified to output each course's title in addition to its enrollment
- The sumEnrollment and findHighestEnrollment modules work only on the course enrollments and there is nothing that they need to do with course titles at all, so, they require no modification. However, we have now gained the ability to say not only what the highest enrollment is, but which one of our courses has that highest enrollment. Let us see below how we might write some code in the mainline logic that utilizes the parallel arrays and the fact that the findHighestEnrollment module actually returns the *index at which it finds the highest enrollment*. What is shown below is not the complete mainline logic, but partial code showing only the parts relevant to this point:

```

integer highestEnrollmentIndex
highestEnrollmentIndex = findHighestEnrollment(enrollment,
MAXIMUM_COURSES)
output "The course with the highest enrollment is " +

```

```

+           title[highestEnrollmentIndex] + " with an enrollment of "
           enrollment[highestEnrollmentIndex]

```

Since we know that the arrays are parallel, i.e., if a course's title is at some index in the title array, then its enrollment will be at an identical index in the enrollment array, once we have the index of the highest enrolling course, we can apply that index to both arrays as shown above, to show both details of that course. If we had a third detail, such as the course number, we could apply the same index to the array containing those too.

Array algorithm 6: Searching an array

Suppose we want to allow the user to enter a course title that they want to find out if we currently offer, and get more details about that course, this would be an operation that requires us to search our title arrays for the desired title.

The algorithm that we learn in this subsection is called the **Linear Search** or **Sequential Search** algorithm.

The item that is being searched for, in this case, the course title that the user wants to find, is called the **search key**.

The Linear Search algorithm simply takes the search key and compares it with every single item in the array to see if they match. We use the same loop again as we have in the previous algorithms that we learned, but with a different couple of preliminary steps, as summarized below:

1. Start at index 0 as before.
2. Compare the search key with the array element at each array index.
3. The search has to stop when we either find a match, in which case the search succeeded (i.e., we offer the course for which the user searched) or when we reach the end of the array, in which case the search failed (i.e., we do not offer the course for which they searched). The latter stopping is already built in – our loop already exits when we reach the end of the array – so, we only have to add the part that stops the search when it succeeds. For this, we declare a Boolean variable that we can set to true when the search succeeds, and modify our loop such that it will execute so long as the search has not succeeded AND we have not reached the end of the array – whichever one of these happens first will terminate the search.
4. If we do find a match, we also copy the index at which we found it, so we can return that index and then the mainline logic can do whatever it needs to in order to inform the user that their search succeeded and give them more details.

Here, then, is the `findTitle` module that applies the Linear Search algorithm. Since it only has to operate on the title array, its parameters are an array containing course titles, the title we want to search for, and the number of courses we offer. It returns the index at which we found

the desired title, or -1 (which is an invalid array index) if the search fails – this is how it signals to the mainline logic whether the search succeeded or failed.

```
integer findTitle(string title[], string desiredTitle,
                  integer coursesOffered)

    integer courseNumber
    boolean isFound
    integer indexOfMatch

    isFound = false
    indexOfMatch = -1
    courseNumber = 0
    while courseNumber < coursesOffered && isFound = false
        if title[courseNumber] = desiredTitle
            isFound = true
            indexOfMatch = courseNumber
        else
            courseNumber = courseNumber + 1
        endif
    endwhile
    return indexOfMatch
endmodule
```

In the above code, you can see that

- `isFound` is initialized to `false`, and the loop executes so long as it remains `false` and we have not reached the end of the array. We only set `isFound` to `true` when we happen to find an array element that equals the search key.
- `indexOfMatch` is initialized to `-1`, and it is only assigned a different value (the index at which we found a match) when we happen to find an array element that equals the search key.

As we did in the previous subsection, here is partial code from the mainline logic that would utilize the `findTitle` module to perform a search:

```
string desiredTitle
```

```

integer matchIndex
output "You can search for courses by title."
output "Please enter a course title to search for: "
input desiredTitle
matchIndex = findTitle(title, desiredTitle, MAXIMUM_COURSES)
output "You searched for the title " + desiredTitle
if matchIndex >= 0
    output "We offer a course with that title!"
    output "Its current enrollment is: " + enrollment[matchIndex]
else
    output "Sorry, we do not offer a course with that title."
endif

```

Since we know that the `findTitle` module will return a valid array index (0 or a positive number) when the search succeeds and -1 when the search fails, we write an `if` statement that tests what that module returns and then show output accordingly.

Array algorithm 7: Sorting an array

Sorting is an operation that is done in several places. You may be looking to order a product online, and after you search for it, sort the search results in order of price, from lowest to highest. In that case, you were deliberately choosing a sorting order. Think of a time when you viewed a list of files on your computer's hard drive or some other disk – the files are often sorted in alphabetic order of name by default, but you can choose to sort them by date modified or by size, or by other criteria.

In our example program in this chapter, we may want to sort our courses in alphabetic order of the course title, or in increasing order of enrollment.

Since sorting is such a heavily used operation, and the amount of data being sorted is often large, it is a task that is computationally intensive, i.e., it can take up a lot of the CPU's processing time. It has been, therefore, an area of much study, and there are several sorting algorithms.

In this subsection, we will learn one of the basic sorting algorithms that is easy to understand and apply, called the **Bubble Sort** algorithm.

The basic operation of the Bubble Sort algorithm can be stated in one simple sentence – it repeatedly compares neighboring array elements, swaps them if they are not in the desired order, until the array is fully sorted. Let us summarize this as a set of steps as we have done with the previous algorithms before we look at code that performs a Bubble Sort.

Here are the steps, assuming an ascending order sort, which would result in the smallest/lowest/alphabetically first array element at index 0 and the largest/highest/alphabetically last array element at the last index in the array:

1. Start at the end of the array, i.e., at the very last array element.
2. Compare that element with the previous one, i.e., at array index one less than it.
3. If the last array element is less than the previous one, then the two of them are not in ascending order, so, swap them.
4. Move up one step, make the same comparison, and swap if necessary.
5. Repeat steps 1 through 4 until we have compared the element at index 1 with the one at index 0 and swapped them if necessary. This final comparison and swap ensures that the correct value ends up at index 0, and we do not have to take that array element into consideration any longer and can focus on sorting the remainder of the array – index 1 through the end of the array. The correct value has “bubbled its way up” to index 0, just like a gas bubble rising to the top of your cup of soda – hence the name Bubble Sort.
6. Repeat steps 1 through 5, each time stopping one index sooner, because you have placed one more value where it belongs.

From the description of the algorithm, you can see that we have two repetitive processes at work here:

1. Make several passes through the array, starting at the end of the array each time.
2. Within each pass, start at the very last array index, compare neighboring elements and swap if necessary, and keep moving up the array, repeating the comparison and swap operations.

From having learned the control structures earlier, we can see that this would require a pair of nested loops – the one that makes us do a sufficient number of passes on the outside, the one that makes comparisons and swaps on the inside.

This brings us to the question of how to control these loops, i.e., how many passes should we make, and when should we stop making comparisons and swaps during each pass?

Let us consider the number of passes required first. As you can see from the description above, after our first pass through the array, the correct value has bubbled its way up to index 0, but the rest of the array is not necessarily sorted. After our second pass, the first two array elements – the ones at indexes 0 and 1 – contain the correct values, but the rest of the array is not necessarily sorted. Likewise, after the third pass, the first three array elements – at indexes 0, 1 and 2 – contain the correct values. If we extend this observation to some more passes, we can conclude that after 9 passes through our enrollment array, the first nine elements – at indexes 0 through 8 – will contain the correct values, which then leaves only one more value, which is located at index 9 and could not possibly be moved anywhere else, so, that too must be where it correctly belongs and therefore, the array is now fully sorted.

Let us generalize this observation. Our enrollment array of size 10 required 9 passes before we could be certain it was fully sorted. An array of size 100, then, would require 99 passes. In general, an array of size n would require $n - 1$ passes. So, our formula for the number of passes required for an array to be fully sorted would be

number of passes = size of the array – 1

Now for the question of when to stop making comparisons and swaps during each pass. Again, from the illustration, we can see that the pass #1 ends when we compare the array element at index 1 with the one at index 0 and swap if necessary. After that first pass, we know that we have the correct value at index 0 and we no longer have to make that comparison and swap, i.e., we can stop one step sooner on pass #2 – when we compare the element at index 2 with the one at index 1. After the second pass, the correct values are located at indexes 0 and 1, and so, on pass #3, we can stop another step sooner – after comparing the elements at index 3 and 2 and swapping them if necessary. Observing the pattern here, we could say that during pass # n , we should stop when we reach index n . This gives us our formula for the inner loop.

Putting all of this together, we can write the outlines of the two loops as shown below:

```
while pass # < size of array
    index = size of array - 1
    while index >= pass #
        compare and swap neighboring elements
    endwhile
endwhile
```

Before we can write actual code that performs a Bubble Sort, there is one additional item to learn – how to swap two array elements. The algorithm swaps neighboring array elements if they are out of order, i.e., it uses an `if` statement of this kind:

```
if array[current index] < array[current index - 1]
    swap the two array elements
endif
```

To swap the two array elements being compared means that we want the value at current index – 1 to move down and take the place of whatever value is at current index, and vice versa.

Suppose we attempt the first step in this swap like this:

```
array[current index] = array[current index - 1]
```

We would end up simply overwriting whatever is at current index with the value at current index – 1, and end up with two copies of the value at current index – 1, as shown in the figure below, where we attempt to swap the course titles CSC 101 and ENG 101, because they are not in alphabetical order within the array, with ENG 101 being currently located at index 8 and CSC

101 at index 9 – ENG 101 is stricken through to show that it has now been overwritten by CSC 101 after performing the step above.

Table 6.4 Incorrect swap illustration

0	
1	
2	
3	
4	
5	
6	
7	
8	“ENG 101” “CSC 101”
9	“CSC 101”

So, how to correctly swap two array elements? This requires a three step process:

1. Copy one of the two values to be swapped, into another variable
2. Overwrite that value with the other one
3. Overwrite the other value with the copy of the first one

These three steps can be shown in outline as follows:

```
copy = array[current index]
array[current index] = array[current index - 1]
array[current index - 1] = copy
```

Now that we know how to swap array elements, let us write code to sort our course enrollments in ascending order (lowest at index 0 to highest at the last index in the array). While we do this, one additional detail for us to pay attention to is that we also have a parallel array containing course titles, and we do not want to destroy the parallel relationship between the arrays. The solution to this is simple – whenever we swap two enrollments, we also swap the course titles at those same indexes – that way, course titles move with the related enrollments and our arrays remain parallel.

```
void sortByEnrollment(string title[], integer enrollment[],
                     integer coursesOffered)

    integer pass
    integer currentIndex
    string copyTitle
    integer copyEnrollment
```

```

pass = 1
while pass < coursesOffered
    currentIndex = coursesOffered - 1
    while currentIndex >= pass
        if enrollment[currentIndex] <
enrollment[currentIndex - 1]
            copyEnrollment = enrollment[currentIndex]
            enrollment[currentIndex] =
                enrollment[currentIndex - 1]
            enrollment[currentIndex - 1] = copyEnrollment

            copyTitle = title[currentIndex]
            title[currentIndex] = title[currentIndex - 1]
            title[currentIndex - 1] = copyTitle
        endif
        currentIndex = currentIndex - 1
    endwhile
    pass = pass + 1
endwhile
return
endmodule

```

As you can see in the above code, before we enter the inner loop each time, we start over with `currentIndex` set to the last index in the array, and then the inner loop compares the element at `currentIndex` with the one at the previous index, i.e., `currentIndex - 1`, and swaps those if they are out of order. We then repeat the same swap with the matching course titles. After that, we decrement `currentIndex` by 1 on each iteration of the inner loop so that we keep moving up the array, and comparing pairs of neighboring elements and swapping. Meanwhile, the outer loop makes us do enough passes to be sure the array is fully sorted.

Try It

Have students try to write a `sortByTitle` module similar to this one – include an explanation that strings can be compared with `<`, `>` operators, if that has not been included in any earlier chapter.

Array algorithm 8: Working with partially-filled arrays

In all of the sections so far, we have been working with the assumption that the array will be completely filled, i.e., if we declare the title array with a size of 1000, then we will always have 1000 course titles in it. In other words, we will always offer exactly 1000 courses. This, however is not a realistic assumption. Programs that utilize arrays often must account for the fact that the array may not be completely filled. To do this, the approach is to declare the array with some maximum size, e.g., we don't know how many courses we will offer every time, but we know we will not offer *more than* 1000 courses – so, we declare the array with that size, but then in our code, account for the fact that the array may not be filled, i.e., that it will be partially filled.

The way to account for this contains two steps:

1. When storing data in the array(s), e.g., storing user input in it, keep a running count of how many array elements have been filled.
2. Use that running count in all algorithms that operate on that array.

The way we have written our code so far, in a modular way, makes this very easy – all we have to do is revise our code that stores the user's input of the course titles and enrollments to keep a running count of how many courses have been entered, and then when we call our modules, pass them that count instead of the `MAXIMUM_COURSES` constant as we did earlier!

Let us now write a module that will allow the user to enter course titles and enrollments, store them in the two arrays, keep a running count of courses that have been entered, and then return that count. As you can see below, we will have to use a sentinel-controlled loop so that the user can let us know when they have finished entering courses:

```
integer getCourseData(string title[], integer enrollment[],
                    integer maximumCourses)

    const string DONE = "AAA"
    integer coursesEntered
    string nextTitle

    coursesEntered = 0
    output "Please enter the details for each course when prompted."
```

```

    output "Please enter the first course title, or " + DONE +
        " to stop: "
    input nextTitle
    while coursesEntered < MAXIMUM_COURSES && nextTitle != DONE
        title[coursesEntered] = nextTitle
        output "Enrollment: "
        input enrollment[coursesEntered]
        coursesEntered = coursesEntered + 1
        output "Enter the title for course #" + (coursesEntered +
1) +
            ", or " + DONE + " to stop: "
        input nextTitle
    endwhile
    return coursesEntered
endmodule

```

As we have done in earlier subsections, here is partial code from the mainline logic that declares the `MAXIMUM_COURSES` constant and the two arrays, and any other variables relevant to this part, and then calls the `getCourseData` module to get the user's input of the course data.

```

const integer MAXIMUM_COURSES = 10
string title[MAXIMUM_COURSES]
integer enrollment[MAXIMUM_COURSES]
integer coursesOffered
coursesOffered = getCourseData(title, enrollment, MAXIMUM_COURSES)

```

Once that module has completed its task and returned control to the mainline logic, the latter now has a count of the courses we are offering, in the `coursesOffered` variable, and when it calls the other modules that operate on the arrays, it can simply pass that count to them instead of the `MAXIMUM_COURSES` constant, e.g., when calling the `findHighestEnrollment` module, we would do

```

integer highestEnrollmentIndex
highestEnrollmentIndex =
    findHighestEnrollment(enrollment, coursesOffered)

```

```

output "The course with the highest enrollment is " +
      title[highestEnrollmentIndex]+
      " with an enrollment of " +
      enrollment[highestEnrollmentIndex]

```

instead of

```
integer highestEnrollmentIndex
```

```
highestEnrollmentIndex =
```

```
    findHighestEnrollment(enrollment, MAXIMUM_COURSES)
```

```

output "The course with the highest enrollment is " +
      title[highestEnrollmentIndex]+
      " with an enrollment of " +
      enrollment[highestEnrollmentIndex]

```

Chapter 7

Multi-dimensional Arrays

Introduction to the Chapter

In Chapter 6, we learned about arrays – more specifically, one-dimensional arrays. This chapter extends that concept by introducing multi-dimensional arrays.

Learning Outcomes

After reading this chapter, students will be able to:

1. Describe what a multi-dimensional array is.
2. Describe when to use a multi-dimensional array as opposed to a one-dimensional array.
3. Write programs in AJANRO that utilize multi-dimensional arrays.

Section 1: Understanding multi-dimensional arrays

What is a multi-dimensional array?

Chapter 6 introduced the concept of the array – a data structure that allows us to store a collection of data that will be used by our program, in the computer's RAM. It then went on to present one-dimensional arrays, and algorithms that operate on them. We can think of a one-dimensional array as allowing us to represent a single column of data in RAM. But what if we need to store data that belongs in a grid or table, or a 3-dimensional structure? This is where multi-dimensional arrays come in.

Let us consider a few examples of such data:

1. A payroll program for a company with one hundred employees needs to store the hours worked by those employees on each day of the week.
2. A weather application needs to store the average temperature during each month of the year, for the last twenty years.

3. A sales analysis program for a retail store chain needs to store the quarterly sales at each of its five locations in two hundred different cities.

The first example would require a table, with one hundred rows for the one hundred employees, and seven columns for the seven days of the week.

Likewise, the second example would require a table with twenty rows – one for each year – and twelve columns for the months within each year.

The third example would require a three-dimensional structure – two hundred rows for the cities, five columns for the store locations within each city, and four such layers, with each layer representing one quarter of the year.

Just as we learned with one-dimensional arrays in Chapter 6, here too, the entire collection of data has to have the same logical meaning, -- in the first example, we have a collection of hours worked; in the second, a collection of average temperatures, and in the third, we have a collection of sales figures.

Now that we know what a multi-dimensional array is, let us see how to use one in a program. In our example programs in this chapter, we will use two-dimensional arrays and learn how to work with those. At the end of the chapter, we will discuss how we can apply the same principles to work with more than two dimensions.

Declaring a two-dimensional array

In Chapter 6, we learned the syntax for declaring a one-dimensional array:

```
data-type arrayName[size of collection]
```

For a two-dimensional array, the only added requirement is that we specify not a single size, but rather, the number of rows and columns we want in the two-dimensional array, using the following general syntax:

```
data-type arrayName[# of rows][# of columns]
```

For the weather app example mentioned above, declaring an array to store the average temperature for each month of the year for 20 years would look like:

```
floating averageTemperature[20][12]
```

Upon executing the declaration above, in RAM, we end up with 20 rows containing 12 columns each, as shown below.

[illegible]

The syntax for doing the same in a two-dimensional array simply adds the requirement to specify both a row index and a column index, as below:

```
arrayName[row index][column index] = value
```

and

```
input arrayName[row index][column index]
```

Both the row and column indexes begin at zero.

For example, to assign the value 23.5 to the very first element (i.e., the first month of the first year) of our `averageTemperature` array, the code would look like

```
averageTemperature[0][0] = 23.5
```

On the other hand, if we wanted to, say, store the user's input of `averageTemperature` in the second element (i.e., the second month of the first year) of our `averageTemperature` array, our code would be

```
input averageTemperature[0][1]
```

Number of rows and columns and highest row and column indexes

With one-dimensional arrays, we learned that since the array indexes begin at zero, the highest index is 1 less than the size of the array.

In the case of two-dimensional arrays, the same kind of relationship holds for the row and column indexes

Highest row index = number of rows in the array – 1

Highest column index = number of columns in the array – 1

We can use this relationship to calculate the highest row and column indexes in any two-dimensional array if we know the number of rows and columns in the array.

Section 2: Array algorithms

Using loops with two-dimensional arrays

As you are familiar with by now, the true power of arrays comes from our ability to use loops in conjunction with them. With one-dimensional arrays, we used a single loop. Two-dimensional arrays, on the other hand, require a pair of nested loops:

1. An outer loop that moves us from row to row
2. An inner loop that moves us from column to column within the current row

Let us try writing a loop that fills our `averageTemperature` array with the user's input.

While writing this code, we will apply what we learned in Chapter 6, about translating array indexes into numbering that the user would expect (e.g., what the user thinks of as year #1 is actually at row index 0 in our two-dimensional array, and likewise for month numbers and column indexes), and earlier in this chapter about the highest row and column index in an array.

```
integer year
integer month
year = 0
while year < 20
    output "Year #" + (year + 1) + ": "
    output "Enter the average temperature for each month when
prompted."
    month = 0
    while month < 12
        output "Month #" + (month + 1) + ": "
        input averageTemperature[year][month]
        month = month + 1
    endwhile
    year = year + 1
endwhile
```

The outer loop starts out on row 0. Within each row, we start at column 0, and fill in all the columns with data from left to right. Once we have filled the last column (column index 11), the month variable is incremented again, making its value 12, and is therefore no longer < 12, causing us to exit the inner loop. We then increment year, which causes us to operate on the next row of the two-dimensional array.

An important detail to observe is that the line of code that initializes year to 0 is before the outer loop, while the similar line that initializes month to 0 is actually *inside the outer loop* and *before the inner loop*. This ensures that for each row (year) we start over at column (month) 0. Otherwise, we would never execute the inner loop after filling the first row.

Here, then, is a list of two-dimensional array algorithms that we will learn:

1. Filling the array – this one, we just learned above.
2. Outputting contents of the array
3. Calculating a row sum – in our example, this would mean adding up the average temperatures for all the months of one year, because we want to, for instance, calculate the average temperature across all the months of the year. To state this in terms of the

year numbers, we want to calculate the average temperature during year 1, during year 2, during year 3, and so on.

4. Calculating a column sum – in our example, this would mean adding up the average temperatures for a single month across all the years, because we want to, for instance, calculate the average temperature for each month over the number of years we are analyzing. To state this in terms of the months, we want to calculate the average temperature in January over the last 20 years, and likewise for February, March, and the rest of the months.

As we did in Chapter 6, we will see each array algorithm applied in a separate module, all of which belong in one program, whose overall goal is to help us analyze temperatures over the last 20 years. After we implement all of the array algorithms that we are going to learn within separate modules, we will write the mainline logic that calls the individual modules when needed.

To make writing these modules easier, let us assume that the following two constants and the two-dimensional array have been declared at the program level, i.e., outside of all modules and outside of the mainline logic, at the very beginning of the program. Recall from Chapter 5 that program level constants and variables would be available to be used by all modules within the program. This will allow us to write these modules with no parameters at all. The two constants that we declare will represent the number of years for which we are analyzing average monthly temperatures, and the number of months in a year.

```
const integer YEARS = 20
```

```
const integer MONTHS_PER_YEAR = 12
```

```
floating averageTemperature[YEARS][MONTHS_PER_YEAR]
```

Given these declarations, in the code for filling the array that we saw earlier, we would replace the numbers 20 and 12 with the names of the above constants.

At the end of the chapter, we will see another example that shows how to write modules that have two-dimensional array parameters. Since we know how to indicate that a module has an array parameter by placing a pair of square brackets after that parameter's name, it is easy to say how to indicate a two-dimensional array parameter – place two pairs of square brackets after the parameter's name.

Array algorithm 2: Outputting contents of a two-dimensional array

This algorithm and the one for filling a two-dimensional array can be summarized with eight steps that are very similar to the four-step one we saw in Chapter 6 with a one-dimensional array:

1. Start at row index 0
2. Set the column index to 0
3. Output whatever is at the current row and column indexes in the array (or in the case of filling the array, store data at the current row and column indexes in the array)

4. Increment the column index on each loop iteration
5. Stop when we pass the highest column index in the array
6. Increment the row index
7. Repeat steps 2 through 6 above
8. Stop when we pass the highest row index in the array

Writing the complete module that would output all of the average temperatures in our current example:

```
void showAverageTemperatures()
    integer year
    integer month
    year = 0
    while year < YEARS
        output "Year #" + (year + 1) + ": "
        output "Here is the average temperature for each month."
        month = 0
        while month < MONTHS_PER_YEAR
            output "Month #" + (month + 1) + ": " +
                averageTemperature[year][month]
            month = month + 1
        endwhile
        year = year + 1
    endwhile
    return
endmodule
```

As you can see, the only difference between this algorithm and the one that fills the array is that this one shows output whereas the other one stores user input in the array. The loops are identical in both. In fact, it will be the same loop, with some modifications, that will appear in the next algorithm.

Array algorithm 3: Calculating a row sum

This algorithm uses the same pair of nested loops again, but requires that we declare an accumulator and initialize it to 0 before the loops, and then add the element at the current index to the running sum in the accumulator. However, we are not calculating just one total – rather, we are calculating a total for each year, i.e., a separate total for each row of the two-dimensional array. Since we have 20 years, then, we would have 20 totals. How do we go about accumulating 20 totals? Declare an accumulator array of size 20! The return type of this module, then, would be an array as well. How to indicate that a module returns an array is shown below – simply type an empty pair of square brackets as part of the return-type. This algorithm also requires an additional step – at the beginning of the outer loop, initialize the accumulator for the current year to 0:

```
floating[] calculateYearlyTotals()
    floating yearlyTotal[YEARS]
    integer year
    integer month
    year = 0
    while year < YEARS
        yearlyTotal[year] = 0
        month = 0
        while month < MONTHS_PER_YEAR
            yearlyTotal[year] = yearlyTotal[year] +
                                averageTemperature[year][month]
            month = month + 1
        endwhile
        year = year + 1
    endwhile
    return yearlyTotal
endmodule
```

Note that when we write a return statement in this module that returns the array of yearly totals, we simply type the return keyword followed by the name of the array that we want to return. As in Chapter 6, writing this module to calculate and return only the yearly totals makes it more flexible as to how it can be used.

Array algorithm 4: Calculating a column sum

This algorithm uses the same pair of loops as the previous one, but in a different order – the loop for months will be the outer loop, while the one for years will be the inner loop. The reason for this is that we want to move not across each row, but *down each column* in order to calculate the monthly totals. Again, we will not have one single monthly total, but rather, 12 monthly totals – one for each month of the year – which requires us to declare an accumulator array of size 12. All the initializations of the year and month variables, and of the monthlyTotal accumulator for the current month, are similarly flipped.

```
floating[] calculateMonthlyTotals()
    floating monthlyTotal[MONTHS_PER_YEAR]
    integer year
    integer month
    month = 0
    while month < MONTHS_PER_YEAR
        monthlyTotal[month] = 0
        year = 0
        while year < YEARS
            monthlyTotal[month] = monthlyTotal[month] +
                averageTemperature[year][month]
            year = year + 1
        endwhile
        month = month + 1
    endwhile
    return monthlyTotal
endmodule
```

What is the correct order in which to operate on rows and columns?

You can see in the first three algorithms above, that we operate on the two-dimensional array in the row-first-column-second order, i.e., our code enters a row of the array and then stays on that row, processing all of the columns in that row, and then advances to the next row, starts over at column 0 and processes all of the columns in that row and so on, until we finish processing the last row of the array.

This is indeed the correct order in which to work on a two-dimensional array, due to the way such arrays are implemented in the computer's RAM.

However, we may have some special requirement, such as wanting to average the temperatures for each month over the last 20 years in this case, that requires us to process the array in the column-first-row-second order as we did in the fourth algorithm above. Only when we have any such special requirement do we process the array in this way.

Try It:

Applying what we learned in Chapter 6 and this chapter, try writing two modules as follows:

1. `showYearlyTotals` – an array containing yearly totals (the totals that were calculated in the module for Algorithm 3 above) is passed to this module, which outputs all of the yearly totals along with the year number; the module returns nothing
 2. `showMonthlyTotals` – an array containing monthly totals (the totals that were calculated in the module for Algorithm 4 above) is passed to this module, which outputs all of the monthly totals along with the month number; the module returns nothing
-

Writing the mainline logic that calls the above modules

As we did in Chapter 6, here is partial code from the mainline logic that declares arrays to receive the data from the ones returned by the last two modules above, declares variables it will use to calculate the `yearlyAverage` and `monthlyAverage`, and calls all the modules in the correct order (assume that we have written a module named `getAverageTemperatures` that contains the code from the very first algorithm to fill the two-dimensional array with the user's input). Finally, our code here also calculates and outputs the average temperature for each year, and then for each month over the 20 years:

```
floating[] yearlyTotal
floating[] monthlyTotal
floating average
integer year
integer month
getAverageTemperatures()
showAverageTemperatures()
yearlyTotal = calculateYearlyTotals()
monthlyTotal = calculateMonthlyTotals()

year = 0
```

```

while year < YEARS
    average = yearlyTotal[year] / MONTHS_PER_YEAR
    output "The average monthly temperature in year #" +
        (year + 1) + " was: " + average
    year = year + 1
endwhile

month = 0
while month < MONTHS_PER_YEAR
    average = monthlyTotal[month] / YEARS
    output "The average temperature for month #" +
        (month + 1) + " was: " + average
    month = month + 1
endwhile

```

Section 3: Beyond two dimensions

So far in this chapter, we have studied two-dimensional arrays in detail. Let us now take a look at what we have observed about the algorithms for one- and two-dimensional arrays.

In Chapter 6, we worked with one-dimensional arrays, and all of the algorithms required a single loop, with the exception of the sorting algorithm.

In this chapter, our algorithms for two-dimensional arrays each required a pair of nested loops.

A general conclusion we could draw then is: each additional dimension requires an additional nested loop – to work on a three-dimensional array, we would have to write three nested loops; for a four-dimensional array, we would write four nested loops and so on.

These loops would follow the same general order as the ones shown for two-dimensional arrays, i.e., row first, column second, layer third, and so on.

Chapter 8

File Input and Output

Introduction to the Chapter

In all of our preceding chapters, our programs received input only from the user, and showed output directly to the user. In this final chapter, we see how to read input from a file and how to write output to a file.

Learning Outcomes

After reading this chapter, students will be able to:

1. Describe what file input and output are.
2. Describe the steps involved in file input and output..
3. Write programs in AJANRO that perform file input and output.

Section 1: Understanding file input and output

What is file input and output?

File input is the act of a program reading some input data that it needs from a file, while file output refers to the program writing, or saving, data and/or results that it produces, to a file.

Much of your personal experience of using a computer has likely involved using a program such as a word-processor to create a file, and also to open an existing file. In fact, when you wrote programs from earlier chapters in this textbook, you typed them in an editor such as Notepad++ and then saved them in text files, which you could later reopen when you wanted to work on the program some more, or simply read it over again. In this chapter, we will focus on data files, i.e., files that contain data that a program requires.

Programs often require large amounts of data in order to provide the service that they are intended to provide. For example, the `EnrollmentAnalyzer` program that we saw earlier in Chapter 6 requires data about all of our courses – each one’s title and enrollment. The version of that program that we saw in Chapter 6 required that the user input all of these details each time we ran the program. It is easy to see that this would be inconvenient for the user, to have to enter all of that data every time they wanted to use the program to analyze our college’s enrollment. It

would be far better if that data were stored in a file and the program could read it from the file each time it ran. In any such actual application, the program would read data from a database rather than a simple file; however, the underlying need that it addresses is the same – that the program needs to store data somewhere and access it every time it runs, and in this chapter, we will use a simple file that stores this data so that we can learn how to read data from a file, and write data to a file.

Before we see how to read and write files, let us first define what a file is. A file is a collection of bytes representing some specific data, stored on some storage medium, such as a hard disk or a flash drive. For example, a picture file contains data about the picture – the colors of the pixels in the picture, and so on; an audio file contains data about the frequencies of the sounds in that piece of audio, etc.; a text file contains codes for characters – letters of the alphabet, digits, etc. The bytes within any file are interpreted by a program that handles that specific kind of data – a program that can display pictures would read in the bytes in a picture file and display the corresponding picture on your screen, an audio player program would read the bytes in an audio file and interpret them as sounds to be played through your speakers, a text editor such as Notepad would read the bytes in a text file and display the corresponding characters on your screen.

In our case, we will assume that all data that our program needs is stored in a text file. We can edit the contents of the file, such as the course titles and enrollments in our EnrollmentAnalyzer example, using a text editor such as Notepad, and then our program can read that data from the file when it needs to. Similarly, our program can write data to a text file, and we can then open that file with Notepad and read what it contains.

How do we read and write files?

In most programming languages, file input and output are supported by tools that exist in the language's library. These tools perform all of the operations that are needed to access the file on the storage medium, such as a hard disk, on which the file is stored, read in the bytes from that file, and then deliver the data to our program in the form that it needs. Consider the example where a program needs to read in some numbers, such as our course enrollments, from a text file. The text file would contain codes for the numeric characters (digits) that you see on the keyboard, such as '5' or '9'. Codes for numeric characters are not actually numeric quantities, however, e.g., the code for the numeric character '5' is not the same as the actual quantity five represented as a binary number. So, the programming tool that reads in numeric data from a text file for our program would have to read in the code for a numeric character such as '5' and then *translate* that into the actual binary value 5 that can be used in calculations, etc. For a program to receive such numeric data, it has to specifically request that the tool give it numeric data by doing all of the translation or conversion work described above.

In our language, again, we simplify all of this by assuming that if we try to read data from a text file into a numeric variable that we have declared, then that data will arrive in the correct form, as a binary number, without us having to specifically request it in numeric form, but if we try to read file data into a string variable, then the character codes from the file will be delivered as they are.

The specific tools that we will use in our language are:

1. An `inputfile` for reading data from a file
2. An `outputfile` for writing data to a file

Next, let us see how to go about using an `inputfile` and an `outputfile`. In each case, we will see a set of general steps to follow, and then an example that applies the steps.

Reading data from a file using an `inputfile`

General steps:

1. Declare an `inputfile` – this looks just like declaring a variable, but with `inputfile` as the data-type. We are said to be declaring a *file handle*. Here is the general syntax:

```
inputfile fileHandleName
```

For example,

```
inputfile courseDataFile
```

2. Use the file handle to open the file.

General syntax:

```
open fileHandleName "location of the file"
```

For example, assuming that our course data is in a file named `courseData.txt`, saved in the `courses` folder on our computer's `C:` drive,

```
open courseDataFile "C:\courses\courseData.txt"
```

3. The file is now open, so, read the data we need from it – we simply use the `input` keyword here to read from a file as well.

General syntax:

```
input variableName1, variableName2, variableName3... from
fileHandleName
```

For example, if we have declared the variables `courseTitle` and `enrollment`:

```
input courseTitle, enrollment from courseDataFile
```

Note that this makes an assumption about the structure of the `courseData.txt` file – that on a single line within that file, we will have one course's title followed by that course's enrollment, separated by a comma. If we have data for 100 courses in the file, then it will contain 100 such lines. Each such line is said to be a record for one course, and we are said to be reading course records from the file. If we want the file to have this kind of structure, then we have to make sure to create the file with that structure so that it will work correctly with our program.

Later, we will see how to read all of the records from a file, one at a time, and store them in an array if needed.

4. Once we have finished reading all the data that we need from the file, close it.

General syntax:

```
close fileHandleName
```

For example,

```
close courseDataFile
```

Writing data to a file using an output file

The general steps are almost identical to those for reading data from a file:

1. Declare an output file

General syntax:

```
outputfile fileHandleName
```

For example,

```
outputfile enrollmentAnalysisFile
```

2. Use the file handle to open the file.

General syntax:

```
open fileHandleName "location of the file"
```

For example, assuming that we want to write our enrollment analysis results to a file named `enrollmentAnalysis.txt`, saved in the `courses` folder on our computer's C: drive,

```
open enrollmentAnalysisFile
    "C:\courses\ enrollmentAnalysis.txt"
```

3. The file is now open, so, write the data we need to it – we simply use the `output` keyword here to read from a file as well.

General syntax:

```
output variableName1, variableName2, variableName3... to
    fileHandleName
```

For example, if we have declared the variables `lowestEnrollment` and `highestEnrollment`:

```
output lowestEnrollment, highestEnrollment to
```

```
enrollmentAnalysisFile
```

This means that on a single line of the `enrollmentAnalysis.txt` file we want to write the lowest and highest enrollments, separated by a comma. In this case we are creating records in the file.

We could also write each result with an explanation, on its own separate line within the file:

```
output "Lowest enrollment: + lowestEnrollment to
      enrollmentAnalysisFile
```

```
output "Highest enrollment: + highestEnrollment to
      enrollmentAnalysisFile
```

We are the ones who decide what we want to write to the file and in what format, so we do whatever is needed to make that happen!

4. Once we have finished writing all the data that we need to the file, close it.

General syntax:

```
close fileName
```

For example,

```
close enrollmentAnalysisFile
```

Reading all records from a file

Writing data to a file is easier than reading data from a file in one sense: we do not have to be concerned about where the end of the file is – wherever the last line is that we chose to write to a file, that is where the file ends. When reading from a file, however, we have to be able to detect when we have reached the end of the file's contents.

Our program does not directly have to be able to detect the end of the file on its own – the computer's operating system will do that for us – but we have to stop trying to read from the file after the end of the file has been detected. Said differently, and in terms of how we write such programs, more accurately, our program tries to continuously read data from the file *so long as we have not reached the end*.

In our language, as in many professional programming languages, the end of file is denoted using the keyword

```
endoffile
```

A loop that attempts to read data from a file continuously so long as we have not reached `endoffile` would look like

```
while !endoffile
```

Recall what the ! is – it is our “not” operator. So, this loop would be read as “while not endoffile”.

Within such a loop, we would write lines of code that read each line from the file, do any processing, such as calculations, scanning for lowest/highest of something, etc., perhaps output results to the user or to another file, or both, and so on.

When we do reach the end of the file, the loop’s condition will no longer be true, causing us to exit the loop. Once we have exited the loop, we close the file, since we have read all of the data from it and do not expect to be reading anything further from it.

One other problem situation we have to keep in mind, though, is that the file may be empty, so, the very first time we try to read from the file, we will have reached the end already, and will not have any data to process. To account for this, we write such loops in the same way we learned to write sentinel-controlled loops:

1. We do an initial, priming input from the file before the loop
2. If we actually managed to read in some data and have not reached the end of the file already, the loop will execute, where we process that initial record of data that we read from the file
3. At the end of the loop, we attempt to read the next record from the file

This way, if our first attempt to read data from the file fails, we never execute the loop’s code at all. If it succeeds, we process the data that we read, and then repeatedly read records from the file and process that data, until we eventually process the last record from the file, after which our next attempt to read from it will cause us to detect that we have reached the end of the file and the loop will then exit.

Try It:

Write lines of code to store data in the arrays as described in each example below:

<come up with at least five different examples using the arrays from the previous Try It exercise>

File Input and Output example

For an example of file input and output, and to see how to read file input into an array, we will now revise the EnrollmentAnalyzer program from Chapter 6 to read its input from a file and output some results to a file.

```
integer readCourseData(string title[], integer enrollment[],
                      integer maximumCourses)
```



```

inputfile courseDataFile
open courseDataFile "C:\courses\courseData.txt"
integer coursesRead

coursesRead = 0
input title[coursesRead], enrollment[coursesRead]
    from courseDataFile
while !endoffile && coursesRead < MAXIMUM_COURSES
    coursesRead = coursesRead + 1
    input title[coursesRead], enrollment[coursesRead]
        from courseDataFile
endwhile
return coursesRead
endmodule

```

As you can see from the while loop's condition in the module above, when reading data from a file into an array, we have to pay attention to two details –

1. Have we reached the end of the file?
2. Have we reached the end of the array?

Whichever one of those happens first will cause us to exit the loop, so that we neither try to read past the end of the file nor try to insert data past the end of the array.

```

void writeEnrollments(string title[]
    integer enrollment[], integer coursesOffered)
outputfile courseListFile
open courseListFile "C:\courses\courseList.txt"
integer coursesWritten
coursesWritten = 0
output "Complete list of course titles and enrollments."
    to courseListFile
while coursesWritten < coursesOffered

```

```

        output title[coursesWritten] + ": " +
                enrollment[coursesWritten] to courseListFile
        coursesWritten = coursesWritten + 1
    endwhile
    return
endmodule

```

The other modules in that example program perform such operations as finding the highest and lowest enrollment, sorting the enrollments in increasing order, searching for a desired course title, and so on. Since they do not perform any input or output, they would remain unchanged. However, the mainline logic from that program might write the results to a file instead of showing the output to the user, or, as shown in the partial code for the mainline logic below, show the results to the user *and* write the same results to a file to be reviewed later whenever the user wants to. The partial code below shows how we might write the highest enrollment to a file.

```

outputfile enrollmentAnalysisFile
open enrollmentAnalysisFile "C:\courses\enrollmentAnalysis.txt"
integer highestEnrollmentIndex
highestEnrollmentIndex = findHighestEnrollment(enrollment,
MAXIMUM_COURSES)
output "The course with the highest enrollment is " +
        title[highestEnrollmentIndex] + " with an enrollment of "
+
        enrollment[highestEnrollmentIndex]
output "The course with the highest enrollment is " +
        title[highestEnrollmentIndex] + " with an enrollment of "
+
        enrollment[highestEnrollmentIndex] to
enrollmentAnalysisFile

```

The first output statement above simply shows the highest enrolling course's title and its enrollment on the screen. The second output statement writes the same result to a file.

If we declare the file handle named `enrollmentAnalysisFile` at the beginning of the mainline logic, and use it to open the `enrollmentAnalysis.txt` file, then we could leave that file open and write all of our results to that file as shown in the sample above, and at the end of the mainline logic, when we have no further results to write, close the file.